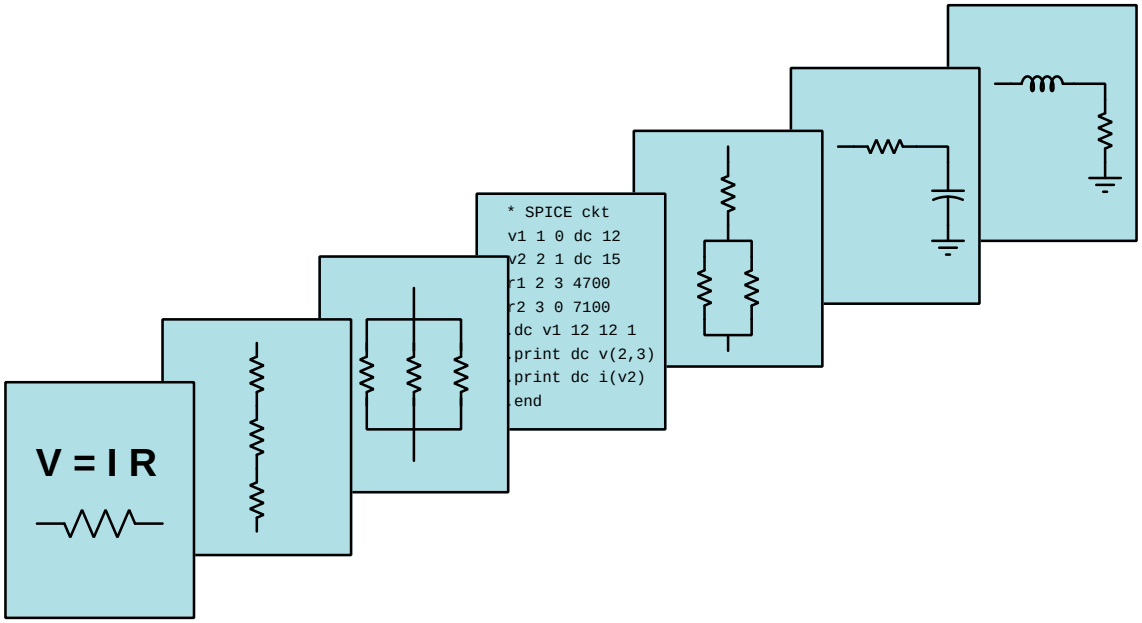
Modular Electronics Learning (ModEL) project



Digital Codes

© 2019-2025 by Tony R. Kuphaldt – under the terms and conditions of the Creative Commons Attribution 4.0 International Public License

Last update = 3 December 2025

This is a copyrighted work, but licensed under the Creative Commons Attribution 4.0 International Public License. A copy of this license is found in the last Appendix of this document. Alternatively, you may visit `http://creativecommons.org/licenses/by/4.0/` or send a letter to Creative Commons: 171 Second Street, Suite 300, San Francisco, California, 94105, USA. The terms and conditions of this license allow for free copying, distribution, and/or modification of all licensed works by the general public.

Contents

1 Introduction **3**
- 1.1 Recommendations for students . . . 3
- 1.2 Challenging concepts related to digital codes . . . 5
- 1.3 Recommendations for instructors . . . 6

2 Tutorial **7**
- 2.1 Morse and Baudot telegraph codes . . . 8
- 2.2 EBCDIC and ASCII codes . . . 9
- 2.3 Hexdumps . . . 11
- 2.4 Unicode . . . 12
- 2.5 UTF-8 encoding examples . . . 13
- 2.6 Binary and Gray position codes . . . 14

3 Historical References **19**
- 3.1 Ancient serial data communication . . . 20
- 3.2 Frank Gray's code . . . 22

4 Derivations and Technical References **31**
- 4.1 International Morse Code . . . 32
- 4.2 ASCII character codes . . . 35

5 Questions **37**
- 5.1 Conceptual reasoning . . . 41
  - 5.1.1 Reading outline and reflections . . . 42
  - 5.1.2 Foundational concepts . . . 43
  - 5.1.3 Morse-coded message . . . 44
  - 5.1.4 ASCII-coded message . . . 44
  - 5.1.5 Gray-binary converters . . . 44
- 5.2 Quantitative reasoning . . . 45
  - 5.2.1 Miscellaneous physical constants . . . 46
  - 5.2.2 Introduction to spreadsheets . . . 47
  - 5.2.3 Using Python to convert between ASCII and hexadecimal . . . 50
  - 5.2.4 Binary and Gray rotary encoders . . . 51
  - 5.2.5 Binary to Gray code conversion . . . 52

5.2.6 Gray code to binary conversion . . . 52
5.3 Diagnostic reasoning . . . 53
5.3.1 Failed ASCII bit . . . 53
5.3.2 Failed encoder photodetector . . . 54

A Problem-Solving Strategies **55**

B Instructional philosophy **57**
B.1 First principles of learning . . . 58
B.2 Proven strategies for instructors . . . 59
B.3 Proven strategies for students . . . 61
B.4 Design of these learning modules . . . 62

C Tools used **65**

D Creative Commons License **69**

E References **77**

F Version history **79**

Index **80**

Chapter 1

Introduction

1.1 Recommendations for students

Binary data is capable of representing far more than just numbers. Combinations of 0 and 1 states are also useful for representing alphabetical characters, machine instructions[1], graphic images[2], audio recordings[3], notes played by musical instruments[4], video recordings[5], and any other information reducible to a finite set of states. This module explores some of the non-numerical applications of binary data.

Important concepts related to digital codes include **binary** counting, how binary states are **represented** in physical form, **standardization** of code patterns, **hexadecimal** representation, **code space**, **code point**, data **compression**, **ASCII**, **Unicode**, **EBCDIC**, **Gray code**, encoding of physical **position**, and **Morse code**.

A recommended strategy for better understanding codes such as ASCII is to take time to manually interpret some of the ASCII code sequences shown in the Tutorial. This takes extra time, but the understanding and confidence you will gain from doing so are worth it. Refer to the ASCII table relating binary values to letters and other characters, then "decode" the ASCII sequences you read in the *hex dump* examples to verify you are properly interpreting them. For Gray code, a good strategy for understanding is to "run" the same thought experiment in your own mind that the Tutorial describes, but with other position states, to see for yourself the purpose and function of that code in comparison to binary. A useful instructional tool is to make yourself some cut-out encoder wheel models using construction paper and use them as experimental tools to explore this same concept in physical form.

[1] Machine instructions include "control characters" to command printers to move to different positions on a paper page, RS274 "G-code" and "M-code" instructions to CNC machine tools commanding the machine go to different positions and different tool speeds, microprocessor instructions telling the CPU to perform arithmetic and logical computations, etc.

[2] Common digital image formats include JPEG, PNG, TIFF, and BMP.

[3] Common digital audio formats include MP3, Ogg Vorbis, and WAV.

[4] The MIDI code (Musical Instrument Digital Interface) describes such note properties as pitch, attack, duration, decay, etc.

[5] Common digital video formats include MPEG, AVI, and WMV.

Here are some good questions to ask of yourself while studying this subject:

- What is the purpose of having digital codes at all?
- What are some historical examples of digital codes, and what do we find in popular use today?
- What information other than alphanumerical characters must a code be able to represent in order to be practically useful?
- How do the Morse and ASCII codes differ from one another?
- How do the ASCII and EBCDIC codes differ from one another?
- Can you identify the ASCII code for any random alphanumeric character you might choose?
- How is "hexdump" useful as a computer utility for the analysis of encoded data?
- Can you translate the ASCII codes (shown in hexadecimal format) into letters from the hexdump example for the Jack London quote?
- What is the rationale for the existence of Unicode?
- What single factor determines the code space for any particular encoding standard?
- What problem was Gray code invented to solve?
- What are some practical applications of a "hex dump" display?
- What is the meaning of "code space"?
- What is the meaning of "code point"?
- How might digitally-encoded text be *compressed* so that fewer bits are necessary to represent the same body of text?
- How does a variable-width coding scheme work?

The *Historical References* chapter contains a fascinating vignette of a military communication code used by ancient Greeks, as well as excerpts from Frank Gray's patent regarding "reflected codes" used to solve a vexing problem with early analog-to-digital converter technology.

1.2 Challenging concepts related to digital codes

The following list cites concepts related to this module's topic that are easily misunderstood, along with suggestions for properly understanding them:

- **Gray versus Binary codes** – binary, like all place-weighted numeration systems, assigns different numerical "weights" to each character's place in the number. Gray code, by contrast, is *not* place-weighted although it is derived from binary by a series of Exclusive-OR (XOR) functions.
- **UTF-8 encoding** – being a *variable-width* encoding scheme, UTF-8 requires certain starting-bit sequences be used to designate the number of bytes that make up a single encoded character, the remaining bits of those successive bytes making up the bits of the Unicode character. The Tutorial section showing UTF-8 encoding examples is most helpful here to see the pattern(s).

1.3 Recommendations for instructors

This section lists realistic student learning outcomes supported by the content of the module as well as suggested means of assessing (measuring) student learning. The outcomes state what learners should be able to do, and the assessments are specific challenges to prove students have learned.

- **Outcome** – Demonstrate effective technical reading and writing

  Assessment – Students present their outlines of this module's instructional chapters (e.g. Case Tutorial, Tutorial, Historical References, etc.) ideally as an entry to a larger Journal document chronicling their learning. These outlines should exhibit good-faith effort at summarizing major concepts explained in the text.

  Assessment – Students show how quantitative results were obtained by the author in the Tutorial chapter's examples.

  Assessment – Students present their "Conceptual Toolbox" Theory Journal entries listing important concepts applied in the Tutorial text(s). A "Conceptual Toolbox" serves as inspiration when solving problems, each concept suggesting a potential problem-solving action. Challenge students to collaboratively generate a Conceptual Toolbox for the day's topic during the class session and then use it when solving problems together, thus fostering the habit of reasoning from principles rather than recalling procedures!

Chapter 2

Tutorial

Binary patterns are not just able to represent numerical values. Given a standardized *code*, they may represent other types of information as well, such as alphabetical characters. The ability to encode non-numerical data in digital form is what allows computers to manipulate and communicate text. In the first few sections we will explore different coding schemes designed to represent written language in digital form, and in the last section we will explore codes used to represent physical positions.

2.1 Morse and Baudot telegraph codes

The first long-distance communication system based on electricity as a signaling medium was the *telegraph*. The discrete electrical state of the telegraph line conductor (current, or no current) might be thought of as equivalent to the 0 and 1 states of a Boolean system. Somehow, people needed to figure out how to represent alphanumeric characters using nothing but on/off states, and from this need *Morse Code* was born.

Morse code represented letters of the alphabet, numerals (0 through 9), and some other characters in the form of "dot" and "dash" signals. Each of these "dot" or "dash" signals represented a change of electrical state in the line, differentiated by time duration, a "dot" being a shorter-duration state than a "dash". In the International Morse Code, no character requires more than five bits of data, and some (such as the common letters E and T) require only one bit.

A table showing how International Morse code represents all characters of the English alphabet and the numerals 0 through 9 appears here:

International Morse Code (English letters and Arabic numerals only)

A •—	J •———	S •••	0 —————
B —•••	K —•—	T —	1 •————
C —•—•	L •—••	U ••—	2 ••———
D —••	M ——	V •••—	3 •••——
E •	N —•	W •——	4 ••••—
F ••—•	O ———	X —••—	5 •••••
G ——•	P •——•	Y —•——	6 —••••
H ••••	Q ——•—	Z ——••	7 ——•••
I ••	R •—•		8 ———••
			9 ————•

The variable bit-length of Morse code, though very efficient[1] in terms of the total number of "dots" and "dashes" required to communicate text messages, was difficult to automate in the form of teletype machines. In answer to this technological problem, Emile Baudot invented a different code where each and every character was five bits in length. Although this gave only 32 characters, which is not enough to represent the 26-letter English alphabet, plus all ten numerals and punctuation symbols, Baudot successfully addressed this problem by designating two of the characters as "shift" characters: one called "letters" and the other called "figures." The other 30 characters had dual (overloaded) meanings, depending on the last "shift" character transmitted[2].

[1] Morse code is an example of a *self-compressing* code, already optimized in terms of minimum bit count. Fixed-field codes such as Baudot and the more modern ASCII tend to waste bandwidth, and may be "compressed" by removing redundant bits.

[2] For example, the Baudot code 11101 meant either "Q" or "1" depending on whether the last shift character was "letters" or "figures," respectively. The code 01010 meant either "R" or "4". The code 00001 meant either "T" or a "5". This overloading of codes is equivalent to using the "shift" key on a computer keyboard to switch between

2.2 EBCDIC and ASCII codes

A much more modern attempt at encoding characters useful for text representation was *EBCDIC*, the "Extended Binary Coded Decimal Interchange Code" invented by IBM in 1962 for use with their line of large ("mainframe") computers. In EBCDIC, each character was represented by a one-byte (eight bit) code, giving this code set 256 (2^8) unique characters. Not only did this provide enough unique characters to represent all the letters of the English alphabet (lower-case *and* capital letters separately!) and numerals 0 through 9, but it also provided a rich set of *control characters* such as "null," "delete," "carriage return," "linefeed," and others useful for controlling the action of electronic printers and other peripheral devices.

A number of EBCDIC codes were unused (or seldom used), though, which made it somewhat inefficient for large data transfers. An attempt to improve this state of affairs was *ASCII*, the "American Standard Code for Information Interchange" first developed in 1963 and then later revised in 1967, both by the American National Standards Institute (ANSI). ASCII is a seven-bit code, one bit shorter per character than EBCDIC, having only 128 unique combinations as opposed to EBCDIC's 256 unique combinations. The compromise made with ASCII versus EBCDIC was a smaller set of control characters.

IBM later created their own "extended" version of ASCII, which was eight bits per character. In this extended code set were included some non-English characters plus special graphic characters, many of which may be placed adjacently on a paper printout or on a computer console display to form larger graphic objects such as lines and boxes.

ASCII is by far the most popular digital code for representing English characters, even today. Nearly every digital transmission of English text in existence employs ASCII as the character encoding[3]. Nearly every text-based computer program's source code is also stored on media using ASCII encoding, where seven-bit codes represent alphanumeric characters comprising the program instructions.

numbers and symbols (e.g. "5" versus "%", or "8" versus "*"). The use of a "shift" key on a keyboard allows single keys on the keyboard to represent multiple characters.

[3] Including the digital source code for this textbook!

The basic seven-bit ASCII code is shown in this table, with the three most significant bits in different columns and the four least significant bits in different rows. For example, the ASCII representation of the upper-case letter "F" is `0b1000110`, the ASCII representation of the equal sign (=) is `0b0111101`, and the ASCII representation of the lower-case letter "q" is `0b1110001`.

ASCII code set

↓ LSB / MSB →	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	–	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

The aforementioned "control characters" occupy the "`000`" and "`001`" columns of the table. These characters, while not associated with a printed character on the page, nevertheless play roles the other "visible" characters cannot. The "LF" character ("line feed"), for example, tells the printer to go to the next line on the paper. The "CR" character ("carriage return") tells the printing head to return to the left-hand side of the paper. The "ESC" character ("escape") does not result in a printed character on paper, nor does it usually command a printing machine to do anything specific, but it is often used within software as a signal to jump up one level in a menu structure, or to switch modes.

2.3 Hexdumps

An interesting and useful way to view ASCII encoding at work in digitally-encoded English sentences is to view a text file (stored in a digital computer) as a series of hexadecimal characters called a *hex dump*. The legacy Unix utility called `hexdump` is easy to use and will be demonstrated here.

First, we will begin with some English text, in this case a quotation by the American author Jack London:

> *The proper function of man is to live, not to exist. I shall not waste my days in trying to prolong them. I shall use my time.*

Saving this text to a file on my computer called `quote.txt`, I then invoke `hexdump` with the following command (typed on the command line of my computer's terminal), using the option `-C` to specify output in the "canonical" form where the displayed text will show the offset[4] values, the hexadecimal bytes, and also an ASCII interpretation of those bytes:

```
hexdump -C quote.txt
```

The resulting output from `hexdump` appears here:

```
00000000  54 68 65 20 70 72 6f 70  65 72 20 66 75 6e 63 74  |The proper funct|
00000010  69 6f 6e 20 6f 66 20 6d  61 6e 20 69 73 20 74 6f  |ion of man is to|
00000020  20 6c 69 76 65 2c 20 6e  6f 74 20 74 6f 20 65 78  | live, not to ex|
00000030  69 73 74 2e 20 20 49 20  73 68 61 6c 6c 20 6e 6f  |ist.  I shall no|
00000040  74 20 77 61 73 74 65 20  6d 79 20 64 61 79 73 20  |t waste my days |
00000050  69 6e 20 74 72 79 69 6e  67 20 74 6f 20 70 72 6f  |in trying to pro|
00000060  6c 6f 6e 67 20 74 68 65  6d 2e 20 20 49 20 73 68  |long them.  I sh|
00000070  61 6c 6c 20 75 73 65 20  6d 79 20 74 69 6d 65 2e  |all use my time.|
00000080  0a                                                |.|
00000081
```

Each ASCII character requires seven bits, but since digital computers are typically organized in bit-widths of powers-of-two (e.g. 4, 8, 16, 32, 64) we find within this text file that eight bits (one byte) are allocated for each 7-bit ASCII character[5]. The left-most column of numbers in this display shows the *offset* value for the start of each line. On the far right is the ASCII interpretation of each byte. Bytes having no alphanumeric meaning appear in this field as a decimal point (`.`) as we see at the very end of the file where the byte `0x0a` (ASCII "linefeed" control character) appears as a dot in the ASCII equivalent field. Note that all spaces in the text are represented by the ASCII code `0x20`. A good instructional exercise is to manually interpret several bytes shown in the hex dump using an ASCII table, verifying the meaning of each one, until one gains confidence in being able to interpret ASCII-encoded text.

[4] The "offset" is simply an integer number expressing how many characters from the beginning the present character resides. The first character, of course, has on offset of zero; the next character an offset of 1, the next character after that an offset of 2, etc.

[5] This makes ASCII text files less efficient than they could be, and for this reason they are easily *compressed* using data-compression algorithms to achieve smaller sizes by eliminating redundant and unused bits.

2.4 Unicode

There exist *many* written languages whose characters cannot and are not represented by either EBCDIC or ASCII. In in attempt to remedy this state of affairs, a new standardized code set exists called *Unicode* utilizing digital words with many more bits than either ASCII or EBCDIC.

When encoding written-language characters as integer-number values, the possible range of integer values available for character encoding is commonly referred to as the *code space*. Any particular integer number within that code space is called a *code point*. Using ASCII as an example, the seven bits provides a code space of one hundred and twenty-eight (2^7), ranging from `0x00` to `0x7F`. The lower-case letter "k" (ASCII code `0b1101011` in binary or `0x6B` in hexadecimal) is one code point within that code space.

In its present version (version 12 as of the year 2019), Unicode supports a code space from `0x0` to `0x10FFFF`, containing 1,114,112 individual code points. The purpose of having such a wide code space is to have the ability (at least in principle) to encode every unique character in every written language on Earth. Not every code point within that code space is mapped to a written-language character. Some characters have multiple code points, for example characters that appear in more than one language alphabet. Also, some characters may be generated by *combinations* of code points (such as certain accent marks appearing in conjunction with other characters). Like ASCII and EBCDIC, Unicode also encodes certain control characters and formatting characters with their own code points (e.g. linefeed).

In deference to existing standards, Unicode encapsulates both ASCII and EBCDIC as sub-sets within its defined code space. For example, the first 128 Unicode characters (`0x000000` through `0x00007F`) are identical to ASCII's 128 characters (`0x00` through `0x7F`).

More than one way exists to package binary bits together to represent Unicode characters in digital systems, each of these methods referred to as a *Unicode Transformation Format* or *UTF*. The simplest method, called UTF-32, allocates 32-bit words for Unicode characters: one 32-bit word per Unicode code point. This method is simple but inefficient because the Unicode code space only requires 21 bits. Two other formats, UTF-16 and UTF-8, use sets of 16-bit and 8-bit digital words to represent Unicode characters. These formats are more memory-efficient but require additional processing to properly interpret (i.e. to "stitch together" multiple words[6] to create the desired Unicode character).

At the time of this writing (2022) UTF-8 is the predominant representation of Unicode for the world-wide web (internet). Both UTF-8 and UTF-16 are known as *variable-width* encoding schemes, meaning certain Unicode characters will be representable with fewer bits than others. Since the first 128 Unicode characters are identical to the ASCII character set, any ASCII characters represented as UTF-8 Unicode are bit-for-bit identical to legacy 7-bit ASCII data words. Unicode characters requiring between 8 and 11 bits are represented in UTF-8 by two bytes (16 bits). Unicode characters requiring 12 to 16 bits result in three-byte (24-bit) UTF-8 words. Finally, Unicode characters requiring more than 16 bits generate four-byte (32-bit) UTF-8 words.

[6] And, *of course*, when communicating such data in serial format there is the matter of big-endian versus little-endian swapping of words!

2.5 UTF-8 encoding examples

Practical examples are helpful for understanding UTF-8 encoding. Let's begin by listing some ASCII characters and their corresponding 1-byte UTF-8 words, these UTF-8 words being nothing more than the seven-bit ASCII code with an extra `0` as "padding" in the most-significant place to round it up to eight bits. The seven "`x`" placeholders shown in the single-byte UTF format are where these ASCII bits go in the UTF-8 word:

- *Single-byte UTF-8 format* = `0xxxxxxx`
- "E" = `0b1000101` ; UTF-8 single-byte encoding = `01000101`
- "+" = `0b0101011` ; UTF-8 single-byte encoding = `00101011`
- (space) = `0b0100000` ; UTF-8 single-byte encoding = `00100000`

Next, we will examine some characters requiring two UTF-8 bytes. Here, the bits comprising the Unicode character must fit into the eleven "`x`" placeholders shown in the double-byte UTF word:

- *Double-byte UTF-8 format* = `110xxxxx 10xxxxxx`
- "Ω" = `0b1110001111` ; UTF-8 double-byte encoding = `11001110 10001111`
- "β" = `0b1110110010` ; UTF-8 double-byte encoding = `11001110 10110010`

Next, we will examine some characters requiring three UTF-8 bytes. Here, the bits comprising the Unicode character must fit into the sixteen "`x`" placeholders shown in the triple-byte UTF word:

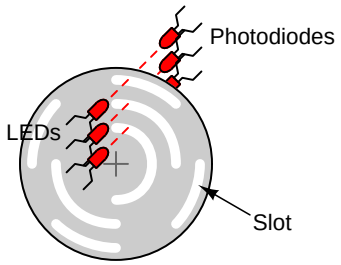
- *Triple-byte UTF-8 format* = `1110xxxx 10xxxxxx 10xxxxxx`
- "†" = `0x2020` = `0b10000000100000` ; UTF-8 triple-byte encoding = `11100010 10000000 10100000`
- "∩" = `0x22C2` = `0b10001011000010` ; UTF-8 triple-byte encoding = `1110010 10001011 10000010`

Lastly, we will examine some characters requiring four UTF-8 bytes, our chosen characters being the digits zero through nine as rendered on legacy 7-segment digital displays, the Unicode hexadecimal words being `0x1FBF0` through `0x1FBF9`, respectively. Here, the bits comprising the Unicode character must fit into the twenty-one "`x`" placeholders shown in the quadruple-byte UTF word:

- *Quadruple-byte UTF-8 format* = `11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`
- 7-segment "0" = `0x1FBF0` = `0b11111101111110000` ; UTF-8 quadruple-byte encoding = `11110000 10011111 10101111 10110000`
- 7-segment "1" = `0x1FBF1` = `0b11111101111110001` ; UTF-8 quadruple-byte encoding = `11110000 10011111 10101111 10110001`

2.6 Binary and Gray position codes

When the physical position of an object must be encoded into a series of binary states, it is possible to use the standard binary (counting) code. For example, consider the wheel of a *rotary encoder* with eight sectors, each sector identified by a unique pattern of slots cut through the wheel allowing light to pass through. The purpose of such a device would be to represent the angular position of a shaft by a digital value, for example the position of a machine component, or the position of a manually-adjusted knob:



3-bit rotary encoder

The three circular rows of slots represent three bits' worth of data, and each of the eight identifiable positions on the wheel corresponds to a three-bit binary value. In the illustration shown above, all three light sources and light detectors have clear view of each other through the slots, and so the binary number `111` would be generated in this position (assuming the successful transmission of light through the wheel corresponds to a "1" state). Clockwise rotation of this wheel into the next sector would result in the binary count "rolling over" from `111` to `000`, and further clockwise rotation would result in the binary count incrementing: `001`, `010`, `011`, etc.

Using standard binary encoding certainly makes sense from the perspective of interfacing a rotary encoder to a digital circuit, because the signals generated are completely conventional and would work directly with decoder circuits, multiplexers, and other digital networks designed to input binary words. However, conventional binary code actually suffers from a major limitation when used to encode the position of an object. At the threshold between successive "count" values where more than one bit changes (for example, from `111` to `000`), anything short of *perfect* alignment between the slots on the encoder wheel will result in false transitional count values.

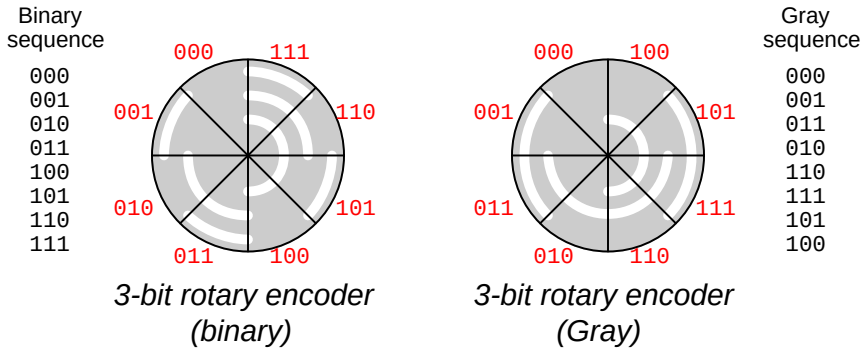
A thought experiment is helpful to picture the problem. Imagine the middle slot on the wheel was cut slightly too far counter-clockwise, so that when the wheel is rotated just past the end of the "seven" sector into the "zero" sector the middle photodetector continues to receive some light after the other two detectors sense darkness. This would result in the count sequence starting at `111`, then briefly transitioning to `010` before settling on `000`. Any circuit receiving this encoder's signal would "think" the shaft instantaneously moved from the "seven" position to the "two" position and then once again made an instantaneous jump to the "zero" position: an impossible feat. This transient error may or may not be significant depending on the application of the encoder, but it is none the less *wrong*. Furthermore, the root cause of such a counting error is the binary coding system itself, which demands absolute perfect physical alignment between the slots to guarantee error-free transitions between all sectors.

A very important point to note here is that these transient counting errors, and the root need for

perfect alignment between the slots cut into the encoder wheel, really only matters at the transitions between count values exhibiting multiple bit changes. Transitioning from 111 to 000 is the worst-case scenario because all three slots would have to end at exactly the same angular position for a clean 111-to-000 transition. However, going from 000 to 001 would not pose quite the same problem if the LSB slot were slightly out of alignment: the transition to 001 might not occur at precisely the right shaft angle, but at least there would be no completely nonsensical counts generated.

An ingenious solution to this problem is to redefine the binary combinations representing each increment in position such that only one bit changes during each transition from one position to the next. The inventor of this technique was a Bell Laboratories researcher named Frank Gray, and such a code is typically referred to as *Gray code* in honor of his innovation. Details of his invention, including instructions on how to determine bit sequences for a word of any bit-length, are presented in the Historical References chapter, section 3.2 beginning on page 22.

Two rotary encoder wheels are shown in the following illustration, the slots in the left-hand wheel representing conventional binary code and the slots in the right-hand wheel representing Gray code. Each of the eight sectors are divided and labeled for ease of interpretation:



Unlike binary, decimal, octal, or hexadecimal, *there are no definite place-weights in Gray code.* That is to say, in Gray code there is no one's place, two's place, four's place, etc. as there is in binary. The number of unique bit combinations for any Gray code is still equal to 2^n (where n is the number of bits), and there is still a definite ordering of bits from most-significant (MSB) to least-significant (LSB) in Gray code, but the similarities end there.

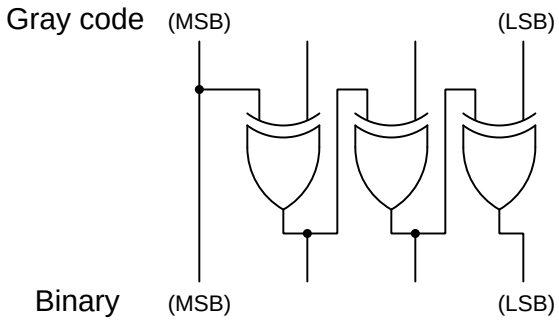
Gray code finds frequent application in position-sensing systems such as rotary and linear encoders, but it is really applicable anywhere slight alignment errors (position errors, timing errors) between parallel bits must be tolerated.

A table comparing a four-bit binary count sequence to Gray code is helpful to reveal patterns:

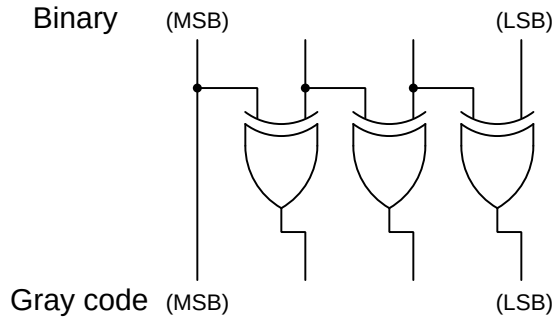
Quantity	**Binary**	**Gray code**
Zero	0000	0000
One	0001	0001
Two	0010	0011
Three	0011	0010
Four	0100	0110
Five	0101	0111
Six	0110	0101
Seven	0111	0100
Eight	1000	1100
Nine	1001	1101
Ten	1010	1111
Eleven	1011	1110
Twelve	1100	1010
Thirteen	1101	1011
Fourteen	1110	1001
Fifteen	1111	1000

If we examine the binary counting sequence from zero to fifteen you will notice how the one's place bit alternates back and forth between 0 and 1, while the two's place bit does the same half as often (e.g. 00110011 rather than 01010101). With each successive place-weight the *frequency* of the bit's alternation is halved. In Gray code, however, the least-signficant place's bit alternates every-other time in the sequence (at the same frequency as the binary code's *two's place* bit), with each successive bit toward the most significant alternating at half the frequency like binary, and with the added twist that each of these alternating-bit patterns begins half-way through the zero states (e.g. 00111100 rather than 00001111).

Four-bit Gray code may be converted into 4-bit binary through the use of Exclusive-OR (XOR) logic functions, as shown below:



Conversely, four-bit binary may be converted into four-bit Gray code by a slightly different network of XOR logic functions as we see here:



An excellent "active reading" exercise is to take any of the corresponding Gray-code and binary code pairs shown in the previous table and see for yourself how these XOR gate networks convert from one to the other by placing the bit-pattern of one code at the inputs and then working through the truth table for each XOR gate to see how the output code develops.

Chapter 3

Historical References

This chapter is where you will find references to historical texts and technologies related to the module's topic.

Readers may wonder why historical references might be included in any modern lesson on a subject. Why dwell on old ideas and obsolete technologies? One answer to this question is that the initial discoveries and early applications of scientific principles typically present those principles in forms that are unusually easy to grasp. Anyone who first discovers a new principle must necessarily do so from a perspective of ignorance (i.e. if you truly *discover* something yourself, it means you must have come to that discovery with no prior knowledge of it and no hints from others knowledgeable in it), and in so doing the discoverer lacks any hindsight or advantage that might have otherwise come from a more advanced perspective. Thus, discoverers are forced to think and express themselves in less-advanced terms, and this often makes their explanations more readily accessible to others who, like the discoverer, comes to this idea with no prior knowledge. Furthermore, early discoverers often faced the daunting challenge of explaining their new and complex ideas to a naturally skeptical scientific community, and this pressure incentivized clear and compelling communication. As James Clerk Maxwell eloquently stated in the Preface to his book *A Treatise on Electricity and Magnetism* written in 1873,

> It is of great advantage to the student of any subject to read the original memoirs on that subject, for science is always most completely assimilated when it is in its nascent state . . . `[page xi]`

Furthermore, grasping the historical context of technological discoveries is important for understanding how science intersects with culture and civilization, which is ever important because new discoveries and new applications of existing discoveries will always continue to impact our lives. One will often find themselves impressed by the ingenuity of previous generations, and by the high degree of refinement to which now-obsolete technologies were once raised. There is much to learn and much inspiration to be drawn from the technological past, and to the inquisitive mind these historical references are treasures waiting to be (re)-discovered.

3.1 Ancient serial data communication

The representation and communication of text by discrete (on/off) signals is rooted in ancient practice. The Greek historian Polybius described one such system in his *Histories* written in the second century BCE, used to communicate information about military maneuvers. In this passage Polybius describes the problem posed by primitive fire signals, and presents an improved method:

> It is evident to all that in every matter, and especially in warfare, the power of acting at the right time contributes very much to the success of enterprises, and fire signals are the most efficient of all the devices that aid us to do this. For they show what has recently occurred and what is still in the course of being done, and by means of them anyone who cares to do so even if he is at a distance of three, four, or even more days' journey can be informed. So that it is always surprising how help can be brought by means of fire messages when the situation requires it. Now in former times, as fire signals were simple beacons, they were for the most part of little use to those who used them. It was possible for those who had agreed to convey a signal that a fleet had arrived in Oreus, Peparethus, or Chalcis, but when it came to some of the citizens having changed sides or having been guilty of treachery or a massacre having taken place in town, or anything of the kind, things that often happen, but cannot all be foreseen – and it is chiefly unexpected occurrences which require instant consideration and help – all such matters defied communication by fire signal. It was quite impossible to have a preconceived code for things which there was no means of foretelling.
>
> This is the vital matter; for how can anyone consider how to render assistance if he does not know how many of the enemy have arrived, or where? And how can anyone be of good cheer or the reverse, or in fact think of anything at all, if he does not understand how many ships or how much corn has arrived from the allies?
>
> The most recent method, devised by Cleoxenus and Democleitus and perfected by myself, is quite definite and capable of dispatching with accuracy every kind of urgent messages, but in practice it requires care and exact attention. It is as follows: We take the alphabet and divide it into five parts, each consisting of five letters. Each of the two parties who signal to each other must get ready five tablets and write one division of the alphabet on each tablet. The dispatcher of the message will raise the first set of torches on the left side indicating which tablet is to be consulted; i.e., one torch if it is the first, two if it is the second, and so on. Next he will raise the second set on the right on the same principle to indicate what letter of the tablet the receiver should write down.

Note the last sentence in the first paragraph: *"It was quite impossible to have a preconceived code for things which there was no means of foretelling."* One could, for example, imagine a fire-signal code consisting of a few distinct patterns of torches, each one encoding a specific military event (e.g. enemy approaching, surrender, victory, retreat, mutiny, riot, type of weapon(s) employed). However, to be of any practical use the number of specific codes would have to be extremely large. Instead, the code described by Polybius neatly handles this problem by leveraging another pre-existing code: *written language*. This way the fire beacons need only encode characters of their alphabet, and with those characters they could describe *any* military event encoded in their written language. The only limitations then would be vocabulary and the speed at which messages could be communicated.

This compounding of codes is very common in digital communication systems. For example, the ASCII code is useful for encoding English alpha-numerical symbols as seven-bit binary words, but when transmitting digital data over long distances we must devise ways to encode the bits themselves as physical signals (e.g. pulses of voltage, current, light, sound, shifting frequencies, etc.). For the sake of argument, we could even employ Polybius' crude fire-torches as the "physical layer" of encoding for an ASCII character set! Once received and decoded, the ASCII symbols could themselves be interpreted as a code representing something other than text. The *Modbus ASCII* standard is a good example of this, where multiple ASCII characters are interpreted to represent "frames" of digital data which could be used to represent any arbitrary information, from numbers to letters to machine instructions to measurements, etc.

3.2 Frank Gray's code

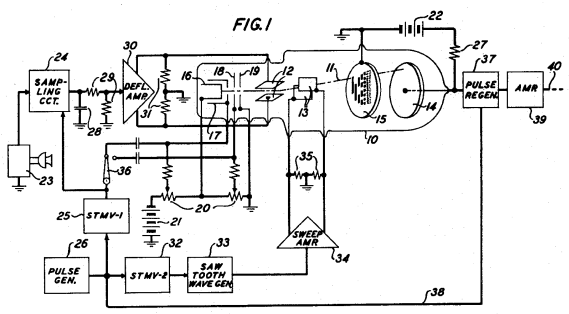
In 1947 a Bell Telephone Laboratories researcher named Frank Gray devised a way to improve a particular form of digital signal communication by replacing standard binary coding with what he called *reflected binary code*. This new code scheme came to be known by his last name, *Gray code*.

To understand Gray's invention, one must grasp the application for which it was invented. Gray's employer, Bell Telephone, designed, built, and maintained long-distance telephone communication networks across the United States. They were interested in converting audio telephone signals into digital form suitable for transmission over long distances with little corruption from external noise and cable attenuation. The digitization of audio telephone signals took the form of voltage pulses following each other in rapid succession, commonly known as a *serial data* stream. Sets of contiguous bits in this data stream represented individually-sampled values of the analog signal.

Gray begins his narrative on page 5 of the patent by explaining how binary coding works:

> This invention relates to pulse code transmission and particularly to the coding of a message signal in a novel code and to the decoding thereof.
>
> In communication by pulse code transmissions the instantaneous amplitudes of a message to be transmitted are successively sampled and each of the successive samples is translated into a code group of on-or-off pulses. By reason of the on-or-off character of the pulses, such a code is denoted a binary code. The number of pulse positions in a code group is the same from group to group. With five such positions the code is a 5-digit binary code. With seven it is a 7-digit binary code; and in general, with in such positions it is an n-digit binary code. A code pulse group of n pulse positions may contain any number, from 0 to n, of "on" pulses. In the conventional n-digit binary code, the number and arrangement of pulses is in accordance with the conventional binary number notation. Thus, for example, with three digits, the number five is written in the conventional binary number notation as 101. Correspondingly, in the conventional 3-digit binary pulse code, the pulses occur in the time sequence P, −, P, where "P" stands for an "on" pulse and "−" stands for an "off" pulse; i. e., a blank pulse position. [page 5]

The technology of that era for converting an analog audio signal into a series of digital pulses used a vacuum tube called a *cathode ray tube*, or CRT. Many different types of CRTs were manufactured, and they all shared the common features of an "electron gun" assembly at one far end generating a thin beam of high-speed electrons, as well as means to bend or deflect this beam before it struck an object at the other end of the evacuated tube. CRTs were once used for television and computer terminal displays, but in this application functioned as a form of *analog to digital converter*. The first page of Gray's patent contains a figure showing the general construction and supporting circuitry of the CRT:



Gray's narrative resumes with a description of the CRT and the analog-to-digital conversion process:

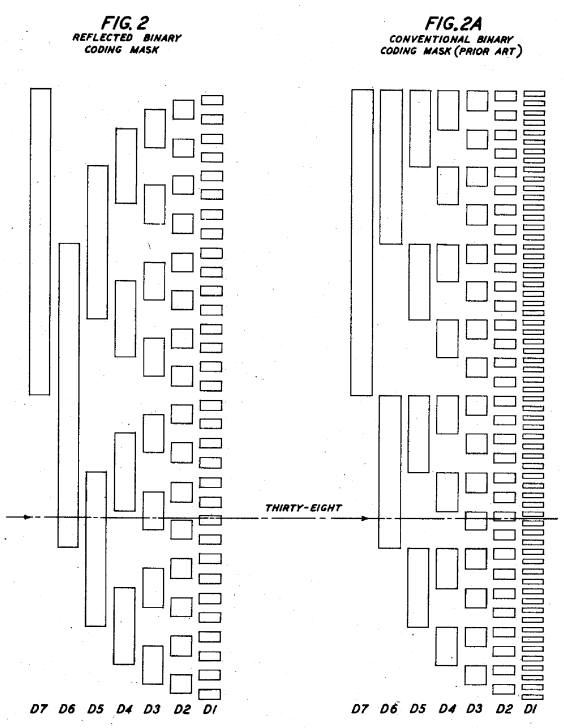
> In Italian Patent 437,300, published June 30, 1948, there is described an instrument for translating message signal samples into code pulse groups in the conventional binary code. In brief, it comprises a cathode beam tube having a coding mask, an electron gun for projecting an electron beam toward the mask, a collector anode for receiving electrons which pass through the mask and deriving pulses from them, means for deflecting the cathode beam in one direction along the mask to a location proportional to the signal sample amplitude, and means for sweeping the beam in a perpendicular direction across the mask between successive sample controlled deflections. As currently employed, the mask comprises a rectangular array of apertures arranged in n columns and 2^n rows, where n is the number of digits of the code. Each aperture row corresponds to a unique value of the signal-controlled beam deflection. The apertures of the various rows are located in conformity with the location of the 1's in a tabulation of successive binary numbers, while the blank portions of the mask are located in conformity with the 0's in the same tabulation. Thus, when the cathode beam is deflected under control of the signal to a particular aperture row and thereupon swept laterally along this row, a train of current pulses may be drawn from the collector whose location on the time scale is in accordance with the arrangement of the 1's and 0's in the binary number whose value is equal to the value of the signal sample being coded. `[page 5]`

The use of a cathode ray tube to convert an analog voltage signal into a serial data stream is rather ingenious[1] and deserves some elaboration. The beam of electrons repeatedly sweeps across the face of the mask in one axis by the direction of a "sawtooth" wave voltage signal applied to a

[1] Not only is this tube design ingenious, but it also stands as an example of just how flexible vacuum tube technology was. Even though vacuum tubes have been made obsolete by semiconductor electronic devices for all but the most

pair of metal plates located to either side of the beam. Another pair of plates perpendicular to the first pair deflects the beam in the other axis, this deflection occurring at the whim of the sampled analog signal. A sample-and-hold circuit (24 in the diagram) captures the analog signal value at the start of the beam sweep and holds that signal voltage constant for the duration of the sweep, so that the beam traces only straight-line paths across the mask. At the conclusion of each sweep, the sample-and-hold circuit re-reads the analog input signal and holds it steady for the next sweep. Holes placed in the mask allow the beam to pass through at certain points where it strikes an anode surface (14) and registers outside the tube as a current pulse. Therefore, as the beam sweeps across a row of the mask, an electrical pulse sequence develops at the anode corresponding to the placement of the holes in that row of the mask.

Figure 2 (from page 2 of the patent) shows two possible mask patterns, one standard binary and the other Gray's "reflected binary" code:



The electron beam's periodic sweep crosses the face of the mask in the horizontal axis (as the masks are shown above), so that the beam "probes" one row at a time. The vertical position of the beam on the mask is determined by the sampled analog signal. Each column of holes on the mask represents one bit (one "digit" as described by Gray) in the binary word. For the sample masks shown, a seven-bit binary sequence is generated for every sweep of the beam from left to right. For an

specialized applications, and for very good reasons (e.g. physical ruggedness, energy efficiency, size, service life), this does not change the fact that this one technology was capable of so many different types of electronic functions. The manipulation of electrons traveling through a vacuum lends itself to a wide range of applications, including amplification, oscillation, rectification, signal modulation, visual display, optical sensing, memory, etc.

analog signal corresponding to the value 38 (shown on the figure), the beam generates the sequence `0100110` as it passes from left to right over that row of holes. Tallying each bit's place-weight, we get 32 + 4 + 2 = 38. A maximum-value analog signal would deflect the beam all the way to the top row where the sequence would be `1111111` (decimal value 127). A minimum-value signal would pull the beam all the way to the bottom to generate `0000001` (decimal value 1). Thus, the sweeping electron beam and the mask together serve as the heart of a seven-bit analog-to-digital converter with a *serial* output.

Gray's narrative continues on page 1, describing the conventional binary coding of the mask shown in Figure 2A. Bear in mind that Gray consistently uses the word "digit" where we would now say "bit":

> It is a characteristic of the conventional binary number notation that a value change of unity may be reflected in the binary number notation by a simultaneous change in several of the digits. Thus, for example, with four digits the number seven is represented by 0111 while the next number, eight, is represented by 1000. In the course of changing the value by one unit, each of the four digits has been changed.
>
> This characteristic of the conventional binary number notation is duplicated in the coding mask of the Italian patent above referred to, and it is a consequence of the resulting arrangement of the apertures that a wander of the beam in the course of its lateral sweep from the correct aperture row to the row immediately above or below it, may result in a coding error which is far greater than the beam deflection error. Various arrangements have been suggested for reducing this coding error by constraining the cathode beam to start its sweep at the correct row and to remain there throughout the sweep . . . All such arrangements involve complexity of apparatus in various degrees.

Gray describes how relatively minor errors in the beam's lateral trajectory ("wander") could cause the beam to graze the holes (or non-holes) of an adjacent row, possibly resulting in a large coding error due to the fact that bit-values often change greatly from one binary integer to the next. To use his example of seven (`0111`) versus eight (`1000`), one can imagine the beam properly deflected to the "eight" row and beginning its left-to-right sweep, successfully passing through the first hole of the "eight" row to create a "1" pulse at the capturing anode, but then a slight downward drift of the beam later in that same sweep might cause it to pass through the three holes of the "seven" row to create three more "1" pulses, the result being a serial data word of `1111` (fifteen) instead of either `1000` (eight) or `0111` (seven). In this example, a vertical alignment "wander" of just 1 "count" results in a digitized error of *seven* counts (i.e. fifteen instead of eight).

Next, Gray begins to describe how his new coding scheme is superior to conventional binary.

> It is a principal object of the present invention to reduce the coding errors in a pulse code transmission system. A more specific object is to provide a pulse code, and a corresponding coding mask, in which the coding error is never greater than the beam deflection error.
>
> Another object is to simplify the manufacture of a coding mask. `[page 5]`
>
> The above objects are attained in accordance with the invention by the selection of a novel form of the binary pulse code which differs from the conventional form by virtue of

a rearrangement of the pulses of the various pulse groups in such a way that the sequence of "on"-pulses [page 5]

and off-pulses which form a pulse group representing a particular signal amplitude differs in only one pulse position from the sequences representing the next lower amplitude and next higher amplitude. The new code is no longer similar to the accepted binary number notation. When it is embodied in a coding mask, the arrangement of the apertures of any one row differs from that of the rows above and below it in not more than one aperture. The resulting mask has certain valuable auxiliary properties and aspects. First, the smallest apertures, that is to say the apertures of the various rows in the column of least digital significance, are twice as large as the apertures of the conventional binary code mask. This makes for ease of manufacture. Second, all of the apertures of the mask, with the sole exception of the single aperture of greatest digital significance, are symmetrically arranged-about a transverse center line. This permits treating the largest digit aperture as an index of polarity only, rectifying the wave to be coded, sampling the rectified wave, and coding the samples using only one half of the coding mask. At the price of some increased complexity of associated apparatus, this greatly reduces the physical dimensions of the coder tube itself. Third, for signals of normal average amplitude range, the beam deflections seldom extend beyond the aperture of the column of second greatest digital significance, so that in the resulting coded signal, the $(n-1)^{th}$ pulse position is nearly always filled. This uniformly filled pulse position affords a convenient source of marker pulses for use in holding a receiver in correct synchronism with the transmitter. [page 6]

Later in the patent (beginning on page 7) the inventor describes his process for developing a "reflected binary" code counting sequence, and in doing so the full meaning of the word "reflected" becomes clear:

> The manner in which the primary reflected binary number system is built up will now be explained.
>
> First: write down the first two numbers in the 1-digit orthodox binary number system, thus:
>
> ```
> Zero     0
> One      1
> ```
>
> Note that the symbols differ in Only one digit. Second: below this array write its "reflection" in a transverse axis:
>
> ```
> Zero     0
> One      1
> ----------
>          1
>          0
> ```
>
> The symbols still differ in not more than one digit. However, the first is identical with the fourth and the second with the third.
>
> Third: to remove this ambiguity, add a second digit to the left of each symbol, 0 for the first two symbols and 1 for the last two, thus:
>
> ```
> Zero    00
> One     01
> Two     11
> Three   10
> ```
>
> and identify the last two symbols with the numbers "two" and "three." Each symbol is now unique and differs from those above and below it in not more than one digit. The array is a representation of the first four numbers in the primary 2-digit reflected binary number system.

The process is next repeated, giving –

First:

```
Zero    00
One     01
Two     11
Three   10
```

Second:

```
Zero    00
One     01
Two     11
Three   10
----------
        10
        11
        01
        00
```

Third:

```
Zero   000
One    001
Two    011
Three  010
Four   110
Five   111
Six    101
Seven  100
```

[page 7]

Gray continues his exposition on this "reflected binary" coding system on page 8 of the patent by describing alternative forms. In every case, though, the coding sequence exhibits the same important property of exhibiting just one bit-change per step, and also the property of all bits being symmetrical about the mid-point except the most-significant bit (MSB).

> Other secondary forms of the reflected binary code may be obtained in various ways. Vertical columns may be interchanged. For any such transposition the pattern may be split along any horizontal division line between rows, and the lower part placed above the upper part, to give a new pattern with the same properties as the primary one. Again, the initial process of building up the code by reflection may be modified, giving two alternatives for the 1-digit code, four for the 2-digit code, and so on. The four alternatives for the 2-digit code are tabulated below. Of these the first is the primary one discussed above, the others being variants.

Number	**Primary**	**First variant**	**Second variant**	**Third variant**
Zero	00	10	01	11
One	01	11	00	10
Two	11	01	10	00
Three	10	00	11	01

[page 8]

Chapter 4

Derivations and Technical References

This chapter is where you will find mathematical derivations too detailed to include in the tutorial, and/or tables and other technical reference material.

4.1 International Morse Code

The following shows a subset of the International Morse Code, the full version containing many non-English characters and other punctuation marks. The full standard for International Morse Code is Recommendation ITU-R M.1677-1 published by the Radiocommunication Sector of the International Telecommunications Union (ITU-R).

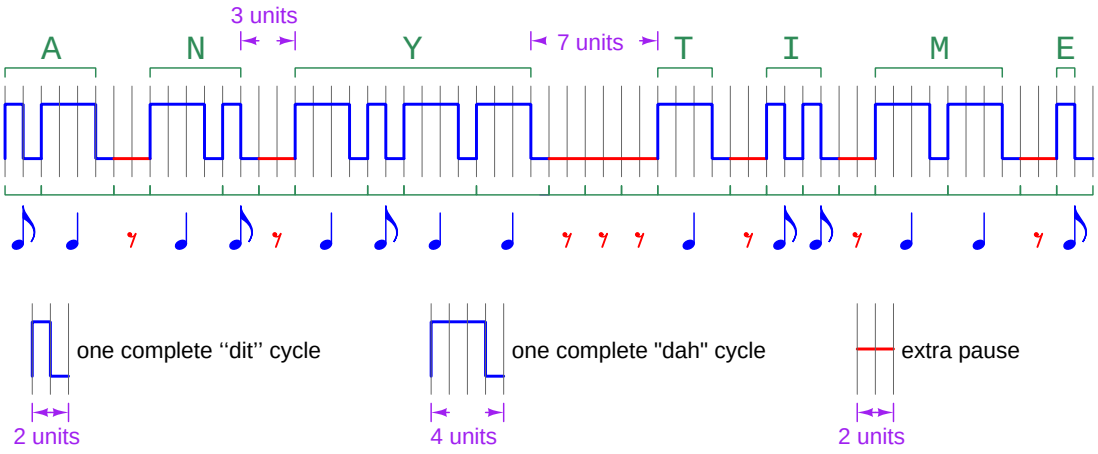
International Morse Code (English letters, Arabic numerals, some symbols)

A •—	J •———	S •••	0 —————
B —•••	K —•—	T —	1 •————
C —•—•	L •—••	U ••—	2 ••———
D —••	M ——	V •••—	3 •••——
E •	N —•	W •——	4 ••••—
F ••—•	O ———	X —••—	5 •••••
G ——•	P •——•	Y —•——	6 —••••
H ••••	Q ——•—	Z ——••	7 ——•••
I ••	R •—•		8 ———••
			9 ————•

? ••——•• *(Question mark)*	, ——••—— *(Comma)*	/ —••—• *(Slash)*	- —••••— *(Hyphen)*
& •—••• *(Ampersand)*	: ———••• *(Colon)*	. •—•—•— *(Period)*	" •—••—• *(Quotation mark)*
× —••— *(Multiplication)*	+ •—•—• *(Addition)*	= —•••— *(Equals)*	•••—•— *(End of work)*
(—•——• *(Left parenthesis)*	) —•——•— *(Right parenthesis)*	′ •————• *(Apostrophe)*	•••••••• *(Backspace delete)*

According to Recommendation ITU-R M.1677 from the ITU Radiocommunication Assembly in 2004, a dash (sounded as "dah") should have a tone duration three times that of a dot (sounded as "dit"), the time delay between dits and dahs within the same character should last as long as the audible duration of a single dit, the time delay between characters should last as long as the audible duration of a dah (three dits), and the space between two words should last as long as the audible duration of seven dits.

If we model each dit as a pulse with a 50% duty cycle (on as long as it is off) and a dah as a pulse with a 75% duty cycle (on three times as long as it is off), we see that the total on/off cycle time of a dah is exactly *twice* the duration of the total on/off cycle time of a dit. As such, we may model dits and dahs and the extra pauses demarcating letters and words using standard musical notations of eighth- and quarter-notes. Consider the following example of the phrase `ANY TIME`, where the smallest subdivision of time (marked by the thin grey vertical lines) is the audible duration of a single dit:



With practice, one's sending of Morse words and phrases will have an almost musical feel. In fact, it is possible to practice sending Morse Code transmissions to the beat of a metronome in order to help learn these precise timing relationships. Getting the pause (rest) durations correct is the most challenging part of this. In real-world practice, if one must compromise on timing accuracy while sending Morse Code, it is better to err on the side of having too much silence between letters and words than too little! The intentional practice of lengthening pause times for easier decoding of individual characters is called *Farnsworth timing*.

Lest anyone think that Morse Code is obsolete, know that this form of communication still enjoys a vibrant following within the amateur radio community for its elegant simplicity as well as for its readability even with low signal-to-noise ratios[1]. Morse Code (or "CW" as amateur radio operators refer to it) is without a doubt the most robust and bandwidth-efficient modulation mode

[1] I have personally experienced the difference between single-sideband telephony (SSB phone) and CW in applications where a voice signal was completely unintelligible but CW pulses sent on the same frequency by the same transmitter were clear and unmistakable. Also, it is routine for low-power (QRP) radio operators to communicate with other people thousands of kilometers away on HF bands using CW mode on transmitters having just 5 Watts of output power, sent through makeshift long-wire antennas!

that human beings are able to directly interpret. Transmitter and receiver hardware for CW radio communication is also far simpler than for any other modulation mode, a crude CW transmitter being little more than an RF oscillator combined with an on/off switch.

If you are interested in learning Morse Code, expect to spend about a week's worth of time learning to send all the alphanumeric characters. Beware that learning to *receive* Morse Code, especially at speed, takes longer than this!

4.2 ASCII character codes

ASCII characters consist of seven-bit digital words. The following table shows all 128 possible combinations of these seven bits, from 0000000 (ASCII "NUL" character) to 1111111 (ASCII "DEL" character). For ease of organization, this table's columns represent the most-significant three bits of the seven-bit word, while the table's rows represent the least-significant four bits. For example, the capital letter "C" would be encoded as 1000011 in the ASCII standard.

↓ LSB / MSB →	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	−	=	M	]	m	}
1110	SO	RS	.	>	N	ˆ	n	˜
1111	SI	US	/	?	O	_	o	DEL

It is worth noting that the ASCII codes for the Arabic numerals 0 through 9 are simply the four-bit binary representation of those numbers preceded by 011. For example, the number six (0110) is represented in ASCII as 0110110; the number three (0011) in ASCII as 0110011; etc. This is useful to know, for example, if you need to program a computer to convert single decimal digits to their corresponding ASCII codes: just take each four-bit numerical value and add forty-eight (0x30 in hexadecimal) to it.

Chapter 5

Questions

This learning module, along with all others in the ModEL collection, is designed to be used in an inverted instructional environment where students independently read[1] the tutorials and attempt to answer questions on their own *prior* to the instructor's interaction with them. In place of lecture[2], the instructor engages with students in Socratic-style dialogue, probing and challenging their understanding of the subject matter through inquiry.

Answers are not provided for questions within this chapter, and this is by design. Solved problems may be found in the Tutorial and Derivation chapters, instead. The goal here is *independence*, and this requires students to be challenged in ways where others cannot think for them. Remember that you always have the tools of *experimentation* and *computer simulation* (e.g. SPICE) to explore concepts!

The following lists contain ideas for Socratic-style questions and challenges. Upon inspection, one will notice a strong theme of *metacognition* within these statements: they are designed to foster a regular habit of examining one's own thoughts as a means toward clearer thinking. As such these sample questions are useful both for instructor-led discussions as well as for self-study.

[1] Technical reading is an essential academic skill for any technical practitioner to possess for the simple reason that the most comprehensive, accurate, and useful information to be found for developing technical competence is in textual form. Technical careers in general are characterized by the need for continuous learning to remain current with standards and technology, and therefore any technical practitioner who cannot read well is handicapped in their professional development. An excellent resource for educators on improving students' reading prowess through intentional effort and strategy is the book textitReading For Understanding – How Reading Apprenticeship Improves Disciplinary Learning in Secondary and College Classrooms by Ruth Schoenbach, Cynthia Greenleaf, and Lynn Murphy.

[2] Lecture is popular as a teaching method because it is easy to implement: any reasonably articulate subject matter expert can talk to students, even with little preparation. However, it is also quite problematic. A good lecture always makes complicated concepts seem easier than they are, which is bad for students because it instills a false sense of confidence in their own understanding; reading and re-articulation requires more cognitive effort and serves to verify comprehension. A culture of teaching-by-lecture fosters a debilitating dependence upon direct personal instruction, whereas the challenges of modern life demand independent and critical thought made possible only by gathering information and perspectives from afar. Information presented in a lecture is ephemeral, easily lost to failures of memory and dictation; text is forever, and may be referenced at any time.

General challenges following tutorial reading

- Summarize as much of the text as you can in one paragraph of your own words. A helpful strategy is to explain ideas as you would for an intelligent child: as simple as you can without compromising too much accuracy.
- Simplify a particular section of the text, for example a paragraph or even a single sentence, so as to capture the same fundamental idea in fewer words.
- Where did the text make the most sense to you? What was it about the text's presentation that made it clear?
- Identify where it might be easy for someone to misunderstand the text, and explain why you think it could be confusing.
- Identify any new concept(s) presented in the text, and explain in your own words.
- Identify any familiar concept(s) such as physical laws or principles applied or referenced in the text.
- Devise a proof of concept experiment demonstrating an important principle, physical law, or technical innovation represented in the text.
- Devise an experiment to disprove a plausible misconception.
- Did the text reveal any misconceptions you might have harbored? If so, describe the misconception(s) and the reason(s) why you now know them to be incorrect.
- Describe any useful problem-solving strategies applied in the text.
- Devise a question of your own to challenge a reader's comprehension of the text.

General follow-up challenges for assigned problems

- Identify where any fundamental laws or principles apply to the solution of this problem, especially before applying any mathematical techniques.
- Devise a thought experiment to explore the characteristics of the problem scenario, applying known laws and principles to mentally model its behavior.
- Describe in detail your own strategy for solving this problem. How did you identify and organized the given information? Did you sketch any diagrams to help frame the problem?
- Is there more than one way to solve this problem? Which method seems best to you?
- Show the work you did in solving this problem, even if the solution is incomplete or incorrect.
- What would you say was the most challenging part of this problem, and why was it so?
- Was any important information missing from the problem which you had to research or recall?
- Was there any extraneous information presented within this problem? If so, what was it and why did it not matter?
- Examine someone else's solution to identify where they applied fundamental laws or principles.
- Simplify the problem from its given form and show how to solve this simpler version of it. Examples include eliminating certain variables or conditions, altering values to simpler (usually whole) numbers, applying a limiting case (i.e. altering a variable to some extreme or ultimate value).
- For quantitative problems, identify the real-world meaning of all intermediate calculations: their units of measurement, where they fit into the scenario at hand. Annotate any diagrams or illustrations with these calculated values.
- For quantitative problems, try approaching it qualitatively instead, thinking in terms of "increase" and "decrease" rather than definite values.
- For qualitative problems, try approaching it quantitatively instead, proposing simple numerical values for the variables.
- Were there any assumptions you made while solving this problem? Would your solution change if one of those assumptions were altered?
- Identify where it would be easy for someone to go astray in attempting to solve this problem.
- Formulate your own problem based on what you learned solving this one.

General follow-up challenges for experiments or projects

- In what way(s) was this experiment or project easy to complete?
- Identify some of the challenges you faced in completing this experiment or project.

- Show how thorough documentation assisted in the completion of this experiment or project.
- Which fundamental laws or principles are key to this system's function?
- Identify any way(s) in which one might obtain false or otherwise misleading measurements from test equipment in this system.
- What will happen if (component X) fails (open/shorted/etc.)?
- What would have to occur to make this system unsafe?

5.1 Conceptual reasoning

These questions are designed to stimulate your analytic and synthetic thinking[3]. In a Socratic discussion with your instructor, the goal is for these questions to prompt an extended dialogue where assumptions are revealed, conclusions are tested, and understanding is sharpened. Your instructor may also pose additional questions based on those assigned, in order to further probe and refine your conceptual understanding.

Questions that follow are presented to challenge and probe your understanding of various concepts presented in the tutorial. These questions are intended to serve as a guide for the Socratic dialogue between yourself and the instructor. Your instructor's task is to ensure you have a sound grasp of these concepts, and the questions contained in this document are merely a means to this end. Your instructor may, at his or her discretion, alter or substitute questions for the benefit of tailoring the discussion to each student's needs. The only absolute requirement is that each student is challenged and assessed at a level equal to or greater than that represented by the documented questions.

It is far more important that you convey your *reasoning* than it is to simply convey a correct answer. For this reason, you should refrain from researching other information sources to answer questions. What matters here is that *you* are doing the thinking. If the answer is incorrect, your instructor will work with you to correct it through proper reasoning. A correct answer without an adequate explanation of how you derived that answer is unacceptable, as it does not aid the learning or assessment process.

You will note a conspicuous lack of answers given for these conceptual questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. The best way by far is to debate the answers with fellow students and also with the instructor during the Socratic dialogue sessions intended to be used with these learning modules. Reasoning through challenging questions with other people is an excellent tool for developing strong reasoning skills.

Another means of checking your conceptual answers, where applicable, is to use circuit simulation software to explore the effects of changes made to circuits. For example, if one of these conceptual questions challenges you to predict the effects of altering some component parameter in a circuit, you may check the validity of your work by simulating that same parameter change within software and seeing if the results agree.

[3] *Analytical* thinking involves the "disassembly" of an idea into its constituent parts, analogous to dissection. *Synthetic* thinking involves the "assembly" of a new idea comprised of multiple concepts, analogous to construction. Both activities are high-level cognitive skills, extremely important for effective problem-solving, necessitating frequent challenge and regular practice to fully develop.

5.1.1 Reading outline and reflections

"Reading maketh a full man; conference a ready man; and writing an exact man" – Francis Bacon

Francis Bacon's advice is a blueprint for effective education: reading provides the learner with knowledge, writing focuses the learner's thoughts, and critical dialogue equips the learner to confidently communicate and apply their learning. Independent acquisition and application of knowledge is a powerful skill, well worth the effort to cultivate. To this end, students should read these educational resources closely, journal their own reflections on the reading, and discuss in detail their findings with classmates and instructor(s). You should be able to do all of the following after reading any instructional text:

☑ Briefly SUMMARIZE THE TEXT in the form of a journal entry documenting your learning as you progress through the course of study. Share this summary in dialogue with your classmates and instructor. Journaling is an excellent self-test of thorough reading because you cannot clearly express what you have not read or did not comprehend.

☑ Demonstrate ACTIVE READING STRATEGIES, including verbalizing your impressions as you read, simplifying long passages to convey the same ideas using fewer words, annotating text and illustrations with your own interpretations, working through mathematical examples shown in the text, cross-referencing passages with relevant illustrations and/or other passages, identifying problem-solving strategies applied by the author, etc. Technical reading is a special case of problem-solving, and so these strategies work precisely because they help solve any problem: paying attention to your own thoughts (metacognition), eliminating unnecessary complexities, identifying what makes sense, paying close attention to details, drawing connections between separated facts, and noting the successful strategies of others.

☑ Identify IMPORTANT THEMES, especially GENERAL LAWS and PRINCIPLES, expounded in the text and express them in the simplest of terms as though you were teaching an intelligent child. This emphasizes connections between related topics and develops your ability to communicate complex ideas to anyone.

☑ Form YOUR OWN QUESTIONS based on the reading, and then pose them to your instructor and classmates for their consideration. Anticipate both correct and incorrect answers, the incorrect answer(s) assuming one or more plausible misconceptions. This helps you view the subject from different perspectives to grasp it more fully.

☑ Devise EXPERIMENTS to test claims presented in the reading, or to disprove misconceptions. Predict possible outcomes of these experiments, and evaluate their meanings: what result(s) would confirm, and what would constitute disproof? Running mental simulations and evaluating results is essential to scientific and diagnostic reasoning.

☑ Specifically identify any points you found CONFUSING. The reason for doing this is to help diagnose misconceptions and overcome barriers to learning.

5.1.2 Foundational concepts

Correct analysis and diagnosis of electric circuits begins with a proper understanding of some basic concepts. The following is a list of some important concepts referenced in this module's full tutorial. Define each of them in your own words, and be prepared to illustrate each of these concepts with a description of a practical example and/or a live demonstration.

Thought experiments as a problem-solving strategy

Morse code

EBCDIC

ASCII

Hexadecimal

Hex dump

Unicode

Code space

Position encoding

Gray code

5.1.3 Morse-coded message

Compose a sentence and translate it into Morse code for your classmates to decode.

Challenges

- Identify modern media over which a Morse-coded message might be conveyed.

5.1.4 ASCII-coded message

Compose a sentence and translate it into ASCII hexadecimal codes for your classmates to decode.

Challenges

- Compare and contrast Morse code against ASCII code communicated serially (i.e. one bit at a time). Is either code better than the other?

5.1.5 Gray-binary converters

Design a circuit to convert four-bit binary to four-bit Gray code. Hint: it uses four XOR gates.

Next, design a circuit to convert four-bit Gray code to four-bit binary. Hint: it also uses four XOR gates!

Challenges

- Describe in detail any design strategies you found helpful in designing the two logic circuits.

5.2 Quantitative reasoning

These questions are designed to stimulate your computational thinking. In a Socratic discussion with your instructor, the goal is for these questions to reveal your mathematical approach(es) to problem-solving so that good technique and sound reasoning may be reinforced. Your instructor may also pose additional questions based on those assigned, in order to observe your problem-solving firsthand.

Mental arithmetic and estimations are strongly encouraged for all calculations, because without these abilities you will be unable to readily detect errors caused by calculator misuse (e.g. keystroke errors).

You will note a conspicuous lack of answers given for these quantitative questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. My advice is to use circuit simulation software such as SPICE to check the correctness of quantitative answers. Refer to those learning modules within this collection focusing on SPICE to see worked examples which you may use directly as practice problems for your own study, and/or as templates you may modify to run your own analyses and generate your own practice problems.

Completely worked example problems found in the Tutorial may also serve as "test cases[4]" for gaining proficiency in the use of circuit simulation software, and then once that proficiency is gained you will never need to rely[5] on an answer key!

[4] In other words, set up the circuit simulation software to analyze the same circuit examples found in the Tutorial. If the simulated results match the answers shown in the Tutorial, it confirms the simulation has properly run. If the simulated results disagree with the Tutorial's answers, something has been set up incorrectly in the simulation software. Using every Tutorial as practice in this way will quickly develop proficiency in the use of circuit simulation software.

[5] This approach is perfectly in keeping with the instructional philosophy of these learning modules: *teaching students to be self-sufficient thinkers*. Answer keys can be useful, but it is even more useful to your long-term success to have a set of tools on hand for checking your own work, because once you have left school and are on your own, there will no longer be "answer keys" available for the problems you will have to solve.

5.2.1 Miscellaneous physical constants

Note: constants shown in **bold** type are *exact*, not approximations. Values inside of parentheses show one standard deviation (σ) of uncertainty in the final digits: for example, the magnetic permeability of free space value given as $1.25663706212(19) \times 10^{-6}$ H/m represents a center value (i.e. the location parameter) of $1.25663706212 \times 10^{-6}$ Henrys per meter with one standard deviation of uncertainty equal to $0.0000000000019 \times 10^{-6}$ Henrys per meter.

Avogadro's number (N_A) = **6.02214076** $\times 10^{23}$ **per mole** (mol^{-1})

Boltzmann's constant (k) = **1.380649** $\times 10^{-23}$ **Joules per Kelvin** (J/K)

Electronic charge (e) = **1.602176634** $\times 10^{-19}$ **Coulomb** (C)

Faraday constant (F) = **96,485.33212...** $\times 10^{4}$ **Coulombs per mole** (C/mol)

Magnetic permeability of free space (μ_0) = $1.25663706212(19) \times 10^{-6}$ Henrys per meter (H/m)

Electric permittivity of free space (ϵ_0) = $8.8541878128(13) \times 10^{-12}$ Farads per meter (F/m)

Characteristic impedance of free space (Z_0) = 376.730313668(57) Ohms (Ω)

Gravitational constant (G) = $6.67430(15) \times 10^{-11}$ cubic meters per kilogram-seconds squared ($m^3/kg\text{-}s^2$)

Molar gas constant (R) = **8.314462618... Joules per mole-Kelvin** (J/mol-K) = 0.08205746(14) liters-atmospheres per mole-Kelvin

Planck constant (h) = **6.62607015** $\times 10^{-34}$ **joule-seconds** (J-s)

Stefan-Boltzmann constant (σ) = **5.670374419...** $\times 10^{-8}$ **Watts per square meter-Kelvin**^{4} ($W/m^2 \cdot K^4$)

Speed of light in a vacuum (c) = **299,792,458 meters per second** (m/s) = 186282.4 miles per second (mi/s)

Note: All constants taken from NIST data "Fundamental Physical Constants – Complete Listing", from `http://physics.nist.gov/constants`, National Institute of Standards and Technology (NIST), 2018 CODATA Adjustment.

5.2.2 Introduction to spreadsheets

A powerful computational tool you are encouraged to use in your work is a *spreadsheet*. Available on most personal computers (e.g. Microsoft Excel), *spreadsheet* software performs numerical calculations based on number values and formulae entered into cells of a grid. This grid is typically arranged as lettered columns and numbered rows, with each cell of the grid identified by its column/row coordinates (e.g. cell `B3`, cell `A8`). Each cell may contain a string of text, a number value, or a mathematical formula. The spreadsheet automatically updates the results of all mathematical formulae whenever the entered number values are changed. This means it is possible to set up a spreadsheet to perform a series of calculations on entered data, and those calculations will be re-done by the computer any time the data points are edited in any way.

For example, the following spreadsheet calculates average speed based on entered values of distance traveled and time elapsed:

	A	B	C	D
1	Distance traveled	46.9	Kilometers	
2	Time elapsed	1.18	Hours	
3	Average speed	= B1 / B2	km/h	
4				
5				

Text labels contained in cells `A1` through `A3` and cells `C1` through `C3` exist solely for readability and are not involved in any calculations. Cell `B1` contains a sample distance value while cell `B2` contains a sample time value. The formula for computing speed is contained in cell `B3`. Note how this formula begins with an "equals" symbol (`=`), references the values for distance and speed by lettered column and numbered row coordinates (`B1` and `B2`), and uses a forward slash symbol for division (`/`). The coordinates `B1` and `B2` function as *variables*[6] would in an algebraic formula.

When this spreadsheet is executed, the numerical value `39.74576` will appear in cell `B3` rather than the formula `= B1 / B2`, because 39.74576 is the computed speed value given 46.9 kilometers traveled over a period of 1.18 hours. If a different numerical value for distance is entered into cell `B1` or a different value for time is entered into cell `B2`, cell `B3`'s value will automatically update. All you need to do is set up the given values and any formulae into the spreadsheet, and the computer will do all the calculations for you.

Cell `B3` may be referenced by other formulae in the spreadsheet if desired, since it is a variable just like the given values contained in `B1` and `B2`. This means it is possible to set up an entire chain of calculations, one dependent on the result of another, in order to arrive at a final value. The arrangement of the given data and formulae need not follow any pattern on the grid, which means you may place them anywhere.

[6] Spreadsheets may also provide means to attach text labels to cells for use as variable names (Microsoft Excel simply calls these labels "names"), but for simple spreadsheets such as those shown here it's usually easier just to use the standard coordinate naming for each cell.

Common[7] arithmetic operations available for your use in a spreadsheet include the following:

- Addition (`+`)
- Subtraction (`-`)
- Multiplication (`*`)
- Division (`/`)
- Powers (`^`)
- Square roots (`sqrt()`)
- Logarithms (`ln()` , `log10()`)

Parentheses may be used to ensure[8] proper order of operations within a complex formula. Consider this example of a spreadsheet implementing the *quadratic formula*, used to solve for roots of a polynomial expression in the form of $ax^2 + bx + c$:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

	A	B
1	x_1	= (-B4 + sqrt((B4^2) - (4*B3*B5))) / (2*B3)
2	x_2	= (-B4 - sqrt((B4^2) - (4*B3*B5))) / (2*B3)
3	a =	9
4	b =	5
5	c =	-2

This example is configured to compute roots[9] of the polynomial $9x^2 + 5x - 2$ because the values of 9, 5, and -2 have been inserted into cells `B3`, `B4`, and `B5`, respectively. Once this spreadsheet has been built, though, it may be used to calculate the roots of *any* second-degree polynomial expression simply by entering the new a, b, and c coefficients into cells `B3` through `B5`. The numerical values appearing in cells `B1` and `B2` will be automatically updated by the computer immediately following any changes made to the coefficients.

[7] Modern spreadsheet software offers a bewildering array of mathematical functions you may use in your computations. I recommend you consult the documentation for your particular spreadsheet for information on operations other than those listed here.

[8] Spreadsheet programs, like text-based programming languages, are designed to follow standard order of operations by default. However, my personal preference is to use parentheses even where strictly unnecessary just to make it clear to any other person viewing the formula what the intended order of operations is.

[9] Reviewing some algebra here, a *root* is a value for x that yields an overall value of zero for the polynomial. For this polynomial ($9x^2 + 5x - 2$) the two roots happen to be $x = 0.269381$ and $x = -0.82494$, with these values displayed in cells `B1` and `B2`, respectively upon execution of the spreadsheet.

Alternatively, one could break up the long quadratic formula into smaller pieces like this:

$$y = \sqrt{b^2 - 4ac} \qquad z = 2a$$

$$x = \frac{-b \pm y}{z}$$

	A	B	C
1	x_1	= (-B4 + C1) / C2	= sqrt((B4^2) - (4*B3*B5))
2	x_2	= (-B4 - C1) / C2	= 2*B3
3	a =	9	
4	b =	5	
5	c =	-2	

Note how the square-root term (y) is calculated in cell C1, and the denominator term (z) in cell C2. This makes the two final formulae (in cells B1 and B2) simpler to interpret. The positioning of all these cells on the grid is completely arbitrary[10] – all that matters is that they properly reference each other in the formulae.

Spreadsheets are particularly useful for situations where the same set of calculations representing a circuit or other system must be repeated for different initial conditions. The power of a spreadsheet is that it automates what would otherwise be a tedious set of calculations. One specific application of this is to simulate the effects of various components within a circuit failing with abnormal values (e.g. a shorted resistor simulated by making its value nearly zero; an open resistor simulated by making its value extremely large). Another application is analyzing the behavior of a circuit design given new components that are out of specification, and/or aging components experiencing drift over time.

[10] My personal preference is to locate all the "given" data in the upper-left cells of the spreadsheet grid (each data point flanked by a sensible name in the cell to the left and units of measurement in the cell to the right as illustrated in the first distance/time spreadsheet example), sometimes coloring them in order to clearly distinguish which cells contain entered data versus which cells contain computed results from formulae. I like to place all formulae in cells below the given data, and try to arrange them in logical order so that anyone examining my spreadsheet will be able to figure out *how* I constructed a solution. This is a general principle I believe all computer programmers should follow: *document and arrange your code to make it easy for other people to learn from it.*

5.2.3 Using Python to convert between ASCII and hexadecimal

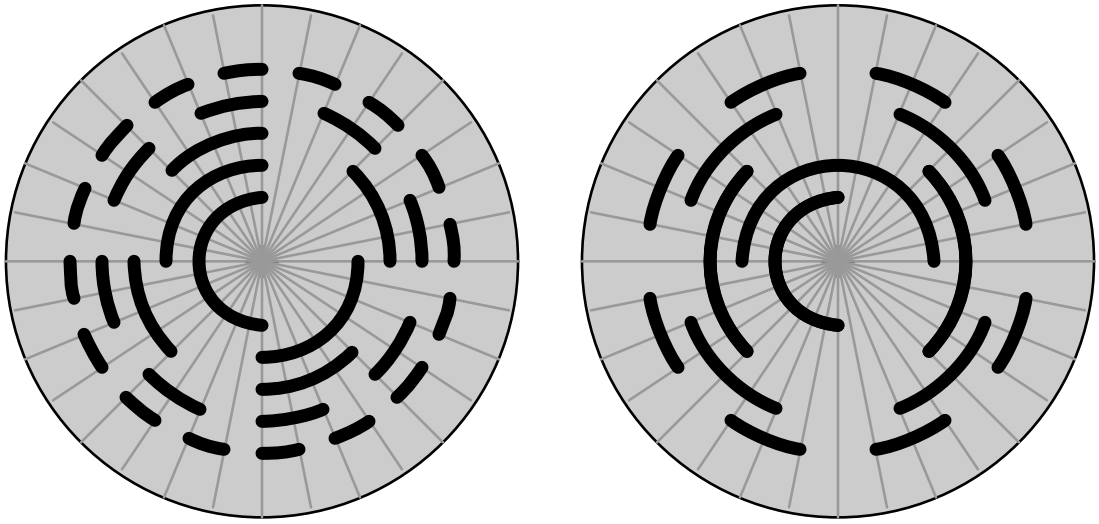
The Python programming language provides two functions useful for converting between ASCII characters and their corresponding numerical codes, `chr()` and `ord()`. Research how to use each of these functions, and then demonstrate their use within a Python interpreter application.

Challenges

- Use the appropriate function to encode a short sentence into a series of hexadecimal values corresponding to the ASCII code.

5.2.4 Binary and Gray rotary encoders

Absolute rotary encoders often use a code known as *Gray code* rather than binary, to represent angular position. This code was patented by Frank Gray of Bell Labs in 1953, as a means of reducing errors in rotary encoder output. Examine each of these encoder disks, and determine which one is binary and which one is Gray code:



Assuming that the darkest areas on the illustration represent slots cut through the disk, and the grey areas represent parts of the disk that are opaque, mark the "zero," "one," and "two" sectors on each disk.

Challenges

- Gray code was originally named "reflected binary code" by its inventor, Frank Gray. Explain why this label makes logical sense by appealing to the slot locations on the Gray code disk.

5.2.5 Binary to Gray code conversion

Convert the following binary numbers into Gray code:

- 100110_2 =
- 110010_2 =
- 101001_2 =
- 1010100110_2 =

Challenges

- Describe a practical application of Gray code, and why it might hold advantages over binary coding.

5.2.6 Gray code to binary conversion

Convert the following binary numbers into Gray code:

- 111110_{Gray} =
- 100001_{Gray} =
- 101110_{Gray} =
- 1110001111_{Gray} =

Challenges

- Describe a practical application where a converter circuit might be necessary to convert Gray code into binary.

5.3 Diagnostic reasoning

These questions are designed to stimulate your deductive and inductive thinking, where you must apply general principles to specific scenarios (deductive) and also derive conclusions about the failed circuit from specific details (inductive). In a Socratic discussion with your instructor, the goal is for these questions to reinforce your recall and use of general circuit principles and also challenge your ability to integrate multiple symptoms into a sensible explanation of what's wrong in a circuit. Your instructor may also pose additional questions based on those assigned, in order to further challenge and sharpen your diagnostic abilities.

As always, your goal is to fully *explain* your analysis of each problem. Simply obtaining a correct answer is not good enough – you must also demonstrate sound reasoning in order to successfully complete the assignment. Your instructor's responsibility is to probe and challenge your understanding of the relevant principles and analytical processes in order to ensure you have a strong foundation upon which to build further understanding.

You will note a conspicuous lack of answers given for these diagnostic questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. The best way by far is to debate the answers with fellow students and also with the instructor during the Socratic dialogue sessions intended to be used with these learning modules. Reasoning through challenging questions with other people is an excellent tool for developing strong reasoning skills.

Another means of checking your diagnostic answers, where applicable, is to use circuit simulation software to explore the effects of faults placed in circuits. For example, if one of these diagnostic questions requires that you predict the effect of an open or a short in a circuit, you may check the validity of your work by simulating that same fault (substituting a very high resistance in place of that component for an open, and substituting a very low resistance for a short) within software and seeing if the results agree.

5.3.1 Failed ASCII bit

Suppose a digital communication system failed in such a way that the MSB of every ASCII code was forced to a "0" state even when it was supposed to be a "1". How would this fault affect the interpretation of the received ASCII characters?

Challenges

- Identify a diagnostic test you could employ to determine that a forced-low MSB was the specific fault.

5.3.2 Failed encoder photodetector

Suppose the photodetector diode for the MSB of a four-bit rotary encoder failed so that it always output a "0" state even when it was supposed to output a "1". Identify how this fault would impact the operation of a four-bit binary-coded encoder versus a four-bit Gray-coded encoder.

Challenges

- Supposing that the successful passage of light through a slot in the encoder wheel represented a "1" state, identify a specific fault that would result in the behavior described.

Appendix A

Problem-Solving Strategies

The ability to solve complex problems is arguably one of the most valuable skills one can possess, and this skill is particularly important in any science-based discipline.

- Study principles, not procedures. Don't be satisfied with merely knowing how to compute solutions – learn *why* those solutions work.
- Identify what it is you need to solve, identify all relevant data, identify all units of measurement, identify any general principles or formulae linking the given information to the solution, and then identify any "missing pieces" to a solution. Annotate all diagrams with this data.
- Sketch a diagram to help visualize the problem. When building a real system, always devise a plan for that system and analyze its function *before* constructing it.
- Follow the units of measurement and meaning of every calculation. If you are ever performing mathematical calculations as part of a problem-solving procedure, and you find yourself unable to apply each and every intermediate result to some aspect of the problem, it means you don't understand what you are doing. Properly done, every mathematical result should have practical meaning for the problem, and not just be an abstract number. You should be able to identify the proper units of measurement for each and every calculated result, and show where that result fits into the problem.
- Perform "thought experiments" to explore the effects of different conditions for theoretical problems. When troubleshooting real systems, perform *diagnostic tests* rather than visually inspecting for faults, the best diagnostic test being the one giving you the most information about the nature and/or location of the fault with the fewest steps.
- Simplify the problem until the solution becomes obvious, and then use that obvious case as a model to follow in solving the more complex version of the problem.
- Check for exceptions to see if your solution is incorrect or incomplete. A good solution will work for *all* known conditions and criteria. A good example of this is the process of testing scientific hypotheses: the task of a scientist is not to find support for a new idea, but rather to *challenge* that new idea to see if it holds up under a battery of tests. The philosophical

principle of *reductio ad absurdum* (i.e. disproving a general idea by finding a specific case where it fails) is useful here.

- Work "backward" from a hypothetical solution to a new set of given conditions.
- Add quantities to problems that are qualitative in nature, because sometimes a little math helps illuminate the scenario.
- Sketch graphs illustrating how variables relate to each other. These may be quantitative (i.e. with realistic number values) or qualitative (i.e. simply showing increases and decreases).
- Treat quantitative problems as qualitative in order to discern the relative magnitudes and/or directions of change of the relevant variables. For example, try determining what happens if a certain variable were to increase or decrease before attempting to precisely calculate quantities: how will each of the dependent variables respond, by increasing, decreasing, or remaining the same as before?
- Consider limiting cases. This works especially well for qualitative problems where you need to determine which direction a variable will change. Take the given condition and magnify that condition to an extreme degree as a way of simplifying the direction of the system's response.
- Check your work. This means regularly testing your conclusions to see if they make sense. This does *not* mean repeating the same steps originally used to obtain the conclusion(s), but rather to use some other means to check validity. Simply repeating procedures often leads to *repeating the same errors* if any were made, which is why alternative paths are better.

Appendix B

Instructional philosophy

B.1 First principles of learning

- **Anyone can learn anything** given appropriate time, effort, resources, challenges, encouragement, and expectations. Dedicating time and investing effort are the student's responsibility; providing resources, challenges, and encouragement are the teacher's responsibility; high expectations are a responsibility shared by both student and teacher.

- **Transfer is not automatic.** The human mind has a natural tendency to compartmentalize information, which means the process of taking knowledge learned in one context and applying it to another usually does not come easy and therefore should never be taken for granted.

- **Learning is iterative.** The human mind rarely learns anything perfectly on the first attempt. Anticipate mistakes and plan for multiple tries to achieve full understanding, using the lessons of those mistakes as feedback to guide future attempts.

- **Information is absorbed, but understanding is created.** Facts and procedures may be memorized easily enough by repeated exposure, but the ability to reliably apply principles to novel scenarios only comes through intense personal effort. This effort is fundamentally creative in nature: casting new concepts in one's own words, running mental experiments to perceive patterns and trends, etc. These acts of making knowledge "one's own" need not be perfect in order to be effective, as the value lies in the activity and not necessarily the finished product. Depth of understanding and permanence of memory are both directly related to mental effort during the learning process – i.e. *easy learning is shallow and fleeting.*

- **Education trumps training.** There is no such thing as an entirely isolated subject, as all fields of knowledge are connected. Training is narrowly-focused and task-oriented. Education is broad-based and principle-oriented. When preparing for a life-long technical career, education beats training every time.

- **Character matters.** Poor habits are more destructive than deficits of knowledge or skill. This is especially true in collective endeavors, where a team's ability to function depends on trust between its members. Simply put, no one wants an untrustworthy person on their team. An essential component of education then, is character development.

- **People learn to be responsible by bearing responsibility.** An irresponsible person is someone who has never *had* to be responsible for anything that mattered enough to them. Just as anyone can learn anything, anyone can become responsible if the personal cost of irresponsibility becomes high enough.

- **What gets measured, gets done.** Accurate and relevant assessment of learning is key to ensuring all students learn. Therefore, it is imperative to measure what matters.

- **Failure is nothing to fear.** Every human being fails, and fails in multiple ways at multiple times. Eventual success only happens when we don't stop trying.

B.2 Proven strategies for instructors

- Assume every student is capable of learning anything they desire given the proper conditions. Treat them as capable adults by granting real responsibility and avoiding artificial incentives such as merit or demerit points.
- Create a consistent culture of high expectations across the entire program of study. Demonstrate and encourage patience, persistence, and a healthy sense of self-skepticism. Anticipate and de-stigmatize error. Teach respect for the capabilities of others as well as respect for one's own fallibility.
- Replace lecture with "inverted" instruction, where students first encounter new concepts through reading and then spend class time in Socratic dialogue with the instructor exploring those concepts and solving problems individually. There is a world of difference between observing someone solve a problem versus actually solving a problem yourself, and so the point of this form of instruction is to place students in a position where they *cannot* passively observe.
- Require students to read extensively, write about what they learn, and dialogue with you and their peers to sharpen their understanding. Apply Francis Bacon's advice that "reading maketh a full man; conference a ready man; and writing an exact man". These are complementary activities helping students expand their confidence and abilities.
- Use artificial intelligence (AI) to challenge student understanding rather than merely provide information. Find productive ways for AI to critique students' clarity of thought and of expression, for example by employing AI as a Socratic-style interlocutor or as a reviewer of students' journals. Properly applied, AI has the ability to expand student access to critical review well outside the bounds of their instructor's reach.
- Build frequent and rapid feedback into the learning process so that students know at all times how well they are learning, to identify problems early and fix them before they grow. Model the intellectual habit of self-assessing and self-correcting your own understanding (i.e. a cognitive *feedback loop*), encouraging students to do the same.
- Use "mastery" as the standard for every assessment, which means the exam or experiment or project must be done with 100% competence in order to pass. Provide students with multiple opportunity for re-tries (different versions of the assessment every time).
- Require students to devise their own hypotheses and procedures on all experiments, so that the process is truly a scientific one. Have students assess their proposed experimental procedures for risk and devise mitigations for those risks. Let nothing be pre-designed about students' experiments other than a stated task (i.e. what principle the experiment shall test) at the start and a set of demonstrable knowledge and skill objectives at the end.
- Have students build as much of their lab equipment as possible: building power sources, building test assemblies[1], and building complete working systems (no kits!). In order to provide

[1] In the program I teach, every student builds their own "Development Board" consisting of a metal chassis with DIN rail, terminal blocks, and an AC-DC power supply of their own making which functions as a portable lab environment they can use at school as well as take home.

this same "ground-up" experience for every new student, this means either previous students take their creations with them, or the systems get disassembled in preparation for the new students, or the systems grow and evolve with each new student group.

- Incorporate external accountability for you and for your students, continuously improving the curriculum and your instructional methods based on proven results. Have students regularly network with active professionals through participation in advisory committee meetings, service projects, tours, jobshadows, internships, etc. Practical suggestions include requiring students to design and build projects for external clients (e.g. community groups, businesses, different departments within the institution), and also requiring students attend all technical advisory committee meetings and to dialogue with the industry representatives at those meetings.
- Repeatedly explore difficult-to-learn concepts across multiple courses, so that students have multiple opportunities to build their understanding.
- Relate all new concepts, whenever possible, to previous concepts and to relevant physical laws. Challenge each and every student, every day, to *reason* from concept to concept and to explain the logical connections between. Challenge students to verify their conclusions by multiple approaches (e.g. double-checking their work using different methods). Ask *"Why?"* often.
- Maintain detailed records on each student's performance and share these records privately with them. These records should include academic performance as well as professionally relevant behavioral tendencies.
- Hold mandatory "check-in" meetings between all program faculty and each new student during their first term. Offer these to all other students as an option, except for any students continuing to manifest unprofessional behaviors, poor academic performance, or who have some other need for a face-to-face meeting with faculty.
- Address problems while they are small, before they grow larger. This is equally true for tutoring technical concepts as it is for helping students build professional habits.
- Build rigorous quality control into the curriculum to ensure every student masters every important concept, and that the mastery is retained over time. This includes (1) review questions added to every exam to re-assess knowledge taught in previous terms, (2) cumulative exams at the end of every term to re-assess all important concepts back to the very beginning of the program, and (3) review assessments in practical (hands-on) coursework to ensure critically-important skills were indeed taught and are still retained. What you will find by doing this is that it actually boosts retention of students by ensuring that important knowledge gets taught and is retained over long spans of time. In the absence of such quality control, student learning and retention tends to be spotty and this contributes to drop-out and failure rates later in their education.
- Finally, *never rush learning.* Education is not a race. Give your students ample time to digest complex ideas, as you continually remind yourself of just how long it took you to achieve mastery! Long-term retention and the consistently correct application of concepts are always the result of *focused effort over long periods of time* which means there are no shortcuts to learning.

B.3 Proven strategies for students

The single most important piece of advice I have for any student of any subject is to take responsibility for your own development in all areas of life including mental development. Expecting others in your life to entirely guide your own development is a recipe for disappointment. This is just as true for students enrolled in formal learning institutions as it is for auto-didacts pursuing learning entirely on their own. Learning to think in new ways is key to being able to gainfully use information, to make informed decisions about your life, and to best serve those you care about. With this in mind, I offer the following advice to students:

- **Approach all learning as valuable**. No matter what course you take, no matter who you learn from, no matter the subject, there is something useful in every learning experience. If you don't see the value of every new experience, you are not looking closely enough!
- **Continually challenge yourself**. Let other people take shortcuts and find easy answers to easy problems. The purpose of education is to stretch your mind, in order to shape it into a more powerful tool. This doesn't come by taking the path of least resistance. An excellent analogy for an empowering education is productive physical exercise: becoming stronger, more flexible, and more persistent only comes through intense personal effort.
- **Master the use of language**. This includes reading extensively, writing every day, listening closely, and speaking articulately. To a great extent language channels and empowers thought, so the better you are at wielding language the better you will be at grasping abstract concepts and articulating them not only for your benefit but for others as well.
- **Do not limit yourself to the resources given to you**. Read books that are not on the reading list. Run experiments that aren't assigned to you. Form study groups outside of class. Take an entrepreneurial approach to your own education, as though it were a business you were building for your future benefit.
- **Express and share what you learn**. Take every opportunity to teach what you have learned to others, as this will not only help them but will also strengthen your own understanding[2].
- Realize that **no one can give you understanding**, just as no one can give you physical fitness. These both must be *built*.
- **Above all, recognize that learning is hard work, and that a certain level of frustration is unavoidable**. There are times when you will struggle to grasp some of these concepts, and that struggle is a natural thing. Take heart that it will yield with persistent and varied[3] effort, and never give up! That concepts don't immediately come to you is not a sign of something wrong, but rather of something right: that you have found a worthy challenge!

[2] On a personal note, I was surprised to learn just how much my own understanding of electronics and related subjects was strengthened by becoming a teacher. When you are tasked every day with helping other people grasp complex topics, it catalyzes your own learning by giving you powerful incentives to study, to articulate your thoughts, and to reflect deeply on the process of learning.

[3] As the old saying goes, "Insanity is trying the same thing over and over again, expecting different results." If you find yourself stumped by something in the text, you should attempt a different approach. Alter the thought experiment, change the mathematical parameters, do whatever you can to see the problem in a slightly different light, and then the solution will often present itself more readily.

B.4 Design of these learning modules

"The unexamined circuit is not worth energizing" – Socrates (if he had taught electricity)

These learning modules, although useful for self-study, were designed to be used in a formal learning environment where a subject-matter expert challenges students to digest the content and exercise their critical thinking abilities in the answering of questions and in the construction and testing of working circuits. Every effort has been made to embed the following instructional and assessment philosophies within:

- The first goal of education is to enhance clear and independent thought, in order that every student reach their fullest potential in a highly complex and inter-dependent world. Robust reasoning is *always* more important than particulars of any subject matter, because its application is universal.
- Literacy is fundamental to independent learning and thought because text continues to be the most efficient way to communicate complex ideas over space and time. Those who cannot read with ease are limited in their ability to acquire knowledge and perspective.
- Articulate communication is fundamental to work that is complex and interdisciplinary.
- Faulty assumptions and poor reasoning are best corrected through challenge, not presentation. The rhetorical technique of *reductio ad absurdum* (disproving an assertion by exposing an absurdity) works well to discipline student's minds, not only to correct the problem at hand but also to learn how to detect and correct future errors.
- Important principles should be repeatedly explored and widely applied throughout a course of study, not only to reinforce their importance and help ensure their mastery, but also to showcase the interconnectedness and utility of knowledge.

These learning modules were expressly designed to be used in an "inverted" teaching environment[4] where students first read the introductory and tutorial chapters on their own, then individually attempt to answer the questions and construct working circuits according to the experiment and project guidelines. The instructor never lectures, but instead meets regularly with each individual student to review their progress, answer questions, identify misconceptions, and challenge the student to new depths of understanding through further questioning. Regular meetings between instructor and student should resemble a Socratic[5] dialogue, where questions serve as scalpels to dissect topics and expose assumptions. The student passes each module only after consistently demonstrating their ability to logically analyze and correctly apply all major concepts in each question or project/experiment. The instructor must be vigilant in probing each student's understanding to ensure they are truly *reasoning* and not just *memorizing*. This is why "Challenge" points appear throughout, as prompts for students to think deeper about topics and as starting points for instructor queries. Sometimes these challenge points require additional knowledge that hasn't been covered in the series to answer in full. This is okay, as the major purpose of the Challenges is to stimulate analysis and synthesis on the part of each student.

The instructor must possess enough mastery of the subject matter and awareness of students' reasoning to generate their own follow-up questions to practically any student response. Even completely correct answers given by the student should be challenged by the instructor for the purpose of having students practice articulating their thoughts and defending their reasoning. Conceptual errors committed by the student should be exposed and corrected not by direct instruction, but rather by reducing the errors to an absurdity[6] through well-chosen questions and thought experiments posed by the instructor. Becoming proficient at this style of instruction requires time and dedication, but the positive effects on critical thinking for both student and instructor are spectacular.

An inspection of these learning modules reveals certain unique characteristics. One of these is a bias toward thorough explanations in the tutorial chapters. Without a live instructor to explain concepts and applications to students, the text itself must fulfill this role. This philosophy results in lengthier explanations than what you might typically find in a textbook, each step of the reasoning process fully explained, including footnotes addressing common questions and concerns students raise while learning these concepts. Each tutorial seeks to not only explain each major concept in sufficient detail, but also to explain the logic of each concept and how each may be developed

[4] In a traditional teaching environment, students first encounter new information via *lecture* from an expert, and then independently apply that information via *homework*. In an "inverted" course of study, students first encounter new information via *homework*, and then independently apply that information under the scrutiny of an expert. The expert's role in lecture is to simply *explain*, but the expert's role in an inverted session is to *challenge*, *critique*, and if necessary *explain* where gaps in understanding still exist.

[5] Socrates is a figure in ancient Greek philosophy famous for his unflinching style of questioning. Although he authored no texts, he appears as a character in Plato's many writings. The essence of Socratic philosophy is to leave no question unexamined and no point of view unchallenged. While purists may argue a topic such as electric circuits is too narrow for a true Socratic-style dialogue, I would argue that the essential thought processes involved with scientific reasoning on *any* topic are not far removed from the Socratic ideal, and that students of electricity and electronics would do very well to challenge assumptions, pose thought experiments, identify fallacies, and otherwise employ the arsenal of critical thinking skills modeled by Socrates.

[6] This rhetorical technique is known by the Latin phrase *reductio ad absurdum*. The concept is to expose errors by counter-example, since only one solid counter-example is necessary to disprove a universal claim. As an example of this, consider the common misconception among beginning students of electricity that voltage cannot exist without current. One way to apply *reductio ad absurdum* to this statement is to ask how much current passes through a fully-charged battery connected to nothing (i.e. a clear example of voltage existing without current).

from "first principles". Again, this reflects the goal of developing clear and independent thought in students' minds, by showing how clear and logical thought was used to forge each concept. Students benefit from witnessing a model of clear thinking in action, and these tutorials strive to be just that.

Another feature of these learning modules is that they do not treat topics in isolation. Rather, important concepts are introduced early in the series, and appear repeatedly as stepping-stones toward other concepts in subsequent modules. This helps to avoid the "compartmentalization" of knowledge, demonstrating the inter-connectedness of concepts and simultaneously reinforcing them. Each module is fairly complete in itself, reserving the beginning of its tutorial to a review of foundational concepts.

To high standards of education,

Tony R. Kuphaldt

Appendix C

Tools used

I am indebted to the developers of many open-source software applications in the creation of these learning modules. The following is a list of these applications with some commentary on each.

You will notice a theme common to many of these applications: a bias toward *code*. Although I am by no means an expert programmer in any computer language, I understand and appreciate the flexibility offered by code-based applications where the user (you) enters commands into a plain ASCII text file, which the software then reads and processes to create the final output. Code-based computer applications are by their very nature *extensible*, while WYSIWYG (What You See Is What You Get) applications are generally limited to whatever user interface the developer makes for you.

The `GNU/Linux` computer operating system

> There is so much to be said about Linus Torvalds' `Linux` and Richard Stallman's `GNU` project. First, to credit just these two individuals is to fail to do justice to the *mob* of passionate volunteers who contributed to make this amazing software a reality. I first learned of `Linux` back in 1996, and have been using this operating system on my personal computers almost exclusively since then. It is *free*, it is completely *configurable*, and it permits the continued use of highly efficient `Unix` applications and scripting languages (e.g. shell scripts, Makefiles, `sed`, `awk`) developed over many decades. `Linux` not only provided me with a powerful computing platform, but its open design served to inspire my life's work of creating open-source educational resources.

Bram Moolenaar's `Vim` text editor

> Writing code for any code-based computer application requires a *text editor*, which may be thought of as a word processor strictly limited to outputting plain-ASCII text files. Many good text editors exist, and one's choice of text editor seems to be a deeply personal matter within the programming world. I prefer `Vim` because it operates very similarly to `vi` which is ubiquitous on `Unix`/`Linux` operating systems, and because it may be entirely operated via keyboard (i.e. no mouse required) which makes it fast to use.

Donald Knuth's TEX typesetting system

Developed in the late 1970's and early 1980's by computer scientist extraordinaire Donald Knuth to typeset his multi-volume magnum opus *The Art of Computer Programming*, this software allows the production of formatted text for screen-viewing or paper printing, all by writing plain-text code to describe how the formatted text is supposed to appear. TEX is not just a markup language for documents, but it is also a Turing-complete programming language in and of itself, allowing useful algorithms to be created to control the production of documents. Simply put, TEX *is a programmer's approach to word processing*. Since TEX is controlled by code written in a plain-text file, this means anyone may read that plain-text file to see exactly how the document was created. This openness afforded by the code-based nature of TEX makes it relatively easy to learn how other people have created their own TEX documents. By contrast, examining a beautiful document created in a conventional WYSIWYG word processor such as Microsoft `Word` suggests nothing to the reader about *how* that document was created, or what the user might do to create something similar. As Mr. Knuth himself once quipped, conventional word processing applications should be called WYSIAYG (What You See Is *All* You Get).

Leslie Lamport's LATEX extensions to TEX

Like all true programming languages, TEX is inherently extensible. So, years after the release of TEX to the public, Leslie Lamport decided to create a massive extension allowing easier compilation of book-length documents. The result was LATEX, which is the markup language used to create all ModEL module documents. You could say that TEX is to LATEX as `C` is to `C++`. This means it is permissible to use any and all TEX commands within LATEX source code, and it all still works. Some of the features offered by LATEX that would be challenging to implement in TEX include automatic index and table-of-content creation.

Tim Edwards' `Xcircuit` drafting program

This wonderful program is what I use to create all the schematic diagrams and illustrations (but not photographic images or mathematical plots) throughout the ModEL project. It natively outputs PostScript format which is a true vector graphic format (this is why the images do not pixellate when you zoom in for a closer view), and it is so simple to use that I have never had to read the manual! Object libraries are easy to create for `Xcircuit`, being plain-text files using PostScript programming conventions. Over the years I have collected a large set of object libraries useful for drawing electrical and electronic schematics, pictorial diagrams, and other technical illustrations.

`Gimp` graphic image manipulation program

Essentially an open-source clone of Adobe's `PhotoShop`, I use `Gimp` to resize, crop, and convert file formats for all of the photographic images appearing in the ModEL modules. Although `Gimp` does offer its own scripting language (called `Script-Fu`), I have never had occasion to use it. Thus, my utilization of `Gimp` to merely crop, resize, and convert graphic images is akin to using a sword to slice bread.

`SPICE` circuit simulation program

`SPICE` is to circuit analysis as TEX is to document creation: it is a form of markup language designed to describe a certain object to be processed in plain-ASCII text. When the plain-text "source file" is compiled by the software, it outputs the final result. More modern circuit analysis tools certainly exist, but I prefer `SPICE` for the following reasons: it is *free*, it is *fast*, it is *reliable*, and it is a fantastic tool for *teaching* students of electricity and electronics how to write simple code. I happen to use rather old versions of `SPICE`, version 2g6 being my "go to" application when I only require text-based output. `NGSPICE` (version 26), which is based on Berkeley `SPICE` version 3f5, is used when I require graphical output for such things as time-domain waveforms and Bode plots. In all `SPICE` example netlists I strive to use coding conventions compatible with all `SPICE` versions.

Andrew D. Hwang's `ePiX` mathematical visualization programming library

This amazing project is a `C++` library you may link to any `C/C++` code for the purpose of generating PostScript graphic images of mathematical functions. As a completely free and open-source project, it does all the plotting I would otherwise use a Computer Algebra System (CAS) such as `Mathematica` or `Maple` to do. It should be said that `ePiX` is *not* a Computer Algebra System like `Mathematica` or `Maple`, but merely a mathematical *visualization* tool. In other words, it won't determine integrals for you (you'll have to implement that in your own `C/C++` code!), but it can graph the results, and it does so beautifully. What I really admire about `ePiX` is that it is a `C++` programming library, which means it builds on the existing power and toolset available with that programming language. Mr. Hwang could have probably developed his own stand-alone application for mathematical plotting, but by creating a `C++` library to do the same thing he accomplished something much greater.

`gnuplot` mathematical visualization software

Another open-source tool for mathematical visualization is `gnuplot`. Interestingly, this tool is *not* part of Richard Stallman's GNU project, its name being a coincidence. For this reason the authors prefer "gnu" *not* be capitalized at all to avoid confusion. This is a much "lighter-weight" alternative to a spreadsheet for plotting tabular data, and the fact that it easily outputs directly to an X11 console or a file in a number of different graphical formats (including PostScript) is very helpful. I typically set my `gnuplot` output format to default (X11 on my Linux PC) for quick viewing while I'm developing a visualization, then switch to PostScript file export once the visual is ready to include in the document(s) I'm writing. As with my use of `Gimp` to do rudimentary image editing, my use of `gnuplot` only scratches the surface of its capabilities, but the important points are that it's *free* and that it *works well*.

`Python` programming language

Both Python and C++ find extensive use in these modules as instructional aids and exercises, but I'm listing Python here as a *tool* for myself because I use it almost daily as a *calculator*. If you open a Python interpreter console and type `from math import *` you can type mathematical expressions and have it return results just as you would on a hand calculator. Complex-number (i.e. *phasor*) arithmetic is similarly supported if you include the complex-math library (`from cmath import *`). Examples of this are shown in the Programming References chapter (if included) in each module. Of course, being a fully-featured programming language, Python also supports conditionals, loops, and other structures useful for calculation of quantities. Also, running in a console environment where all entries and returned values show as text in a chronologically-ordered list makes it easy to copy-and-paste those calculations to document exactly how they were performed.

Appendix D

Creative Commons License

Creative Commons Attribution 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 – Definitions.

a. **Adapted Material** means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. **Adapter's License** means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. **Copyright and Similar Rights** means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

d. **Effective Technological Measures** means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

e. **Exceptions and Limitations** means fair use, fair dealing, and/or any other exception or

limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

f. **Licensed Material** means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

g. **Licensed Rights** means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

h. **Licensor** means the individual(s) or entity(ies) granting rights under this Public License.

i. **Share** means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

j. **Sui Generis Database Rights** means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

k. **You** means the individual or entity exercising the Licensed Rights under this Public License. **Your** has a corresponding meaning.

Section 2 – Scope.

a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

A. reproduce and Share the Licensed Material, in whole or in part; and

B. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures.

For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

5. Downstream recipients.

A. Offer from the Licensor – Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

B. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. Other rights.

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

2. Patent and trademark rights are not licensed under this Public License.

3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 – License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. Attribution.

1. If You Share the Licensed Material (including in modified form), You must:

A. retain the following if it is supplied by the Licensor with the Licensed Material:

i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);

ii. a copyright notice;

iii. a notice that refers to this Public License;

iv. a notice that refers to the disclaimer of warranties;

v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

B. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

C. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4 – Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 – Disclaimer of Warranties and Limitation of Liability.

a. Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors,

whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.

b. To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 – Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

2. upon express reinstatement by the Licensor.

For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 – Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 – Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully

be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the "Licensor." Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at `creativecommons.org/policies`, Creative Commons does not authorize the use of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at `creativecommons.org`.

Appendix E

References

Gray, Frank, *US Patent 2,632,058*, "Pulse Code Communication", application filed 13 November 1947, patent granted 17 March 1953.

Kramer, Thomas R.; Proctor, Frederick M.; Messina, Elena, *The NIST RS274/NGC Interpreter*, Version 3, NIST Interagency/Internal Report (NISTIR) report number 6556, National Institute of Standards and Technology, Gaithersburg, Maryland, 17 August 2000.

Polybius, *Histories*, cited from pages 49-50 of *Lapham's Quarterly* Volume IX, Number 1, Winter 2016, American Agora Foundation, New York, NY, 2016.

The Unicode Standard, Version 12.0 – Core Specification, The Unicode Consortium, Mountain View, CA, 2019.

Appendix F

Version history

This is a list showing all significant additions, corrections, and other edits made to this learning module. Each entry is referenced by calendar date in reverse chronological order (newest version first), which appears on the front cover of every learning module for easy reference. Any contributors to this open-source document are listed here as well.

3 December 2025 – corrected an error spotted by Wyatt Gleason, where the two logic diagrams showing Binary-to-Gray code conversion and Gray-to-Binary code conversion were swapped.

30 April 2025 – added some instructor notes.

4 December 2024 – corrected a typographical error in the Tutorial, courtesy of Caleb Wing.

9 November 2024 – divided the Introduction chapter into sections, one with recommendations for students, one with a listing of challenging concepts, and one with recommendations for instructors.

28 April 2024 – divided the Tutorial chapter into more sections.

7 December 2022 – added more questions to the Introduction chapter.

8 August 2022 – identified the two Python language functions that were formerly unidentified in the "Using Python to convert between ASCII and hexadecimal" Quantitative Reasoning question. Also added content to the Tutorial on UTF-8 as an encoding scheme for Unicode characters.

4 May 2022 – minor additions to the Introduction, and also made a minor typo correction in the Polybius quote (my copy said "messages" when it should have said "message").

9 December 2021 – added Quantitative Reasoning question prompting students to use the `ord()` and `chr()` built-in Python functions to explore ASCII codes.

9 July 2021 – replaced some TeX-style italicizing markup with LaTeX-style.

9 May 2021 – commented out or deleted empty chapters.

2 December 2020 – minor additions to the Introduction.

30 November 2020 – minor additions to the Introduction.

23 August 2020 – significantly edited the Introduction chapter to make it more suitable as a pre-study guide and to provide cues useful to instructors leading "inverted" teaching sessions.

28 January 2020 – added Foundational Concepts to the list in the Conceptual Reasoning section.

23 December 2019 – minor edits to a question.

16 December 2019 – added some screenshots of "hex dump" output from digital files to show ASCII encoding at work. Also updated the Tutorial on Unicode, since the text I drew from (of my own writing years before) was somewhat out of date.

15 December 2019 – continued writing Tutorial chapter and completed the transcription of relevant portions of Frank Gray's patent to the Historical References chapter.

14 December 2019 – document first created.

Index

Adding quantities to a qualitative problem, 56
Annotating diagrams, 55
ASCII, 9

Baudot code, 8
Bell Telephone, 22
Big-endian, 12

Checking for exceptions, 56
Checking your work, 56
Code point, 12
Code space, 12
Code, computer, 65
Computer, mainframe, 9
Control characters, 9

Data stream, 22
Dimensional analysis, 55

EBCDIC, 9
Edwards, Tim, 66

Farnsworth timing, 33
Fire signals, 20
Frequency, 16

G code, 3
Graph values to solve a problem, 56
Gray code, 15, 22
Gray, Frank, 15, 22
Greenleaf, Cynthia, 37

Hex dump, 11
How to teach with these modules, 63
Hwang, Andrew D., 67

Identify given data, 55
Identify relevant principles, 55
Intermediate results, 55
International Morse Code, 32
International Morse code, 8
Inverted instruction, 63

Knuth, Donald, 66

Lamport, Leslie, 66
Limiting cases, 56
Little-endian, 12
London, Jack, 11

Maxwell, James Clerk, 19
Metacognition, 42
Modbus, 21
Moolenaar, Bram, 65
Morse code, 8
Murphy, Lynn, 37

Open-source, 65

Polybius, Greek historian, 20
Problem-solving: annotate diagrams, 55
Problem-solving: check for exceptions, 56
Problem-solving: checking work, 56
Problem-solving: dimensional analysis, 55
Problem-solving: graph values, 56
Problem-solving: identify given data, 55
Problem-solving: identify relevant principles, 55
Problem-solving: interpret intermediate results, 55
Problem-solving: limiting cases, 56
Problem-solving: qualitative to quantitative, 56
Problem-solving: quantitative to qualitative, 56
Problem-solving: reductio ad absurdum, 56
Problem-solving: simplify the system, 55
Problem-solving: thought experiment, 14, 55
Problem-solving: track units of measurement, 55

Problem-solving: visually represent the system, 55
Problem-solving: work in reverse, 56

Qualitatively approaching a quantitative problem, 56

Reading Apprenticeship, 37
Reductio ad absurdum, 56, 62, 63
Reflected binary code, 22
RS274, 3

Schoenbach, Ruth, 37
Scientific method, 42
Serial digital data, 22
Simplifying a system, 55
Socrates, 62
Socratic dialogue, 63
SPICE, 37
Stallman, Richard, 65

Thought experiment, 14, 55
Torvalds, Linus, 65

Unicode transformation format, 12
Units of measurement, 55
UTF-16, 12
UTF-32, 12
UTF-8, 12

Visualizing a system, 55

Work in reverse to solve a problem, 56
WYSIWYG, 65, 66